Integrating Ontologies with CAR Mappings[⋆]

Manuel Wimmer[1], Horst Kargl[1], Martina Seidl[1], Michael Strommer[1], and Thomas Reiter[2]

[1] Business Informatics Group
Institute of Software Technology and Interactive Systems
Vienna University of Technology, Austria
{wimmer|kargl|seidl|strommer}@big.tuwien.ac.at
[2] Information Systems Group
Johannes Kepler University Linz, Austria
reiter@ifs.uni-linz.ac.at

Abstract. CAR is a declarative language which can be used to map between different schemas in general and which allows the integration of ontologies in special. CAR provides mapping operators whose expressivity exceeds the possibilities of simple equivalence mappings by far. Due to the declarative nature of this approach, CAR which we implemented in a graphical mapping framework, offers a very user-friendly way to overcome schematic heterogenities.
Furthermore, the mappings can be executed on Colored Petri Nets, and therefore they can be easily simulated and debugged in a framework for representing schemas and mapping models as Transformation Nets which can be applied to transform concrete instances of schemas expressed as tokens.

1 Introduction

The integration of heterogenous information resources has a long history in computer science. The problem of finding correspondences between distinct data sources arises in very different fields, e.g. in database research [4, 7, 8], in the XML area [5] and in object-oriented refactoring [1].

Nowadays ontologies are often used as schemas for semantic web applications. Ontologies are often directly generated from database schemas or XML schemas. Unfortunately, heterogeneities between schemas are not eliminated when ontologies are generated and also quite common between manually defined ontologies due to different modeling styles. Furthermore, the different modeling styles do seldom result in models with just trivial differences like distinct names. Thus, for the integration of ontologies simple equivalence mappings are not enough.

Consequently, common integration solutions use imperative or hybrid (declarative and imperativ) languages [3] to define the complex mappings which are

[⋆] This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-810806.0

then hidden in global variables, lookup operations, and user-defined trace models. This solution has two main drawbacks, first, the definition of imperative code is a tedious and error-prone task and second, the actual mappings are hard to discover and to understand. Therefore, we propose a declarative and bidirectional mapping language named CAR, which can be used for the core concepts of object-oriented modeling languages such can be found in UML or OWL.

The paper focuses on the basic mapping operators and how they are used. We consider schematic heterogenities (i.e., the two schemas which will be mapped are in the same language and have (almost) the same semantics but they differ in their structure). We present some mapping problems of concrete ontologies, which are not possible to be solved with simple equivalent mappings. Then we show how they can be mapped with our CAR mapping language. Furthermore, we discuss the interpretation of the mappings in context of translation scenarios and report how the mappings are actually executed with Colored Petri Nets. Our approach yields multiple benefits: (1) an easy to use mapping language, which has a graphical syntax and (2) the execution model based on Colored Petri Nets allows a user-friendly simulation and debugging of the mappings.

2 Motivating Examples

Sometimes simple equivalence mappings (like mapping a class to a class, mapping a relation to a relation, and mapping an attribute to an attribute) are not sufficient to resolve the heterogenties between two ontologies. Consider the examples shown in Figure 1. For the sake of readability we use the UML-class-diagram notation.

Example 1. The first example contains a well-known problem from the database community [4]: a *Person* has a *firstName* and *lastName*. In an alternative modeling approach the person's family could be modeled explicitly by introducing a class *Family* which now contains the *lastName* (under the assumption that all members of a family have the same last name and that the last name of a family is unique). So suddenly we have two classes instead of one. Nevertheless, both approaches express almost the same.

Example 2. The second example is similar to Example 1 but instead of aggregation it deals with inheritance. It illustrates a typical refactoring pattern from object-oriented programming [1], namely the way how an inheritance hierarchy can be flattened. In the context of a university, *Scientific Employees* and *Non-Scientific Employees* specialize *Employees*. Alternatively, an enumeration datatype *Employment* with the values "SE" and "NSE" could be introduced. Finally, the type of employment of an employee can be expressed by adding an attribute of the enumeration type to the class *Employee*.

Example 3. The third example is about inverse relationships. When a family is modeled this can be done by saying "2 persons are the parents of arbitrary many children" or alternatively " a person has 2 persons as parents". So here we do not have any additional classes but the relationship is interpreted differently in the two models.

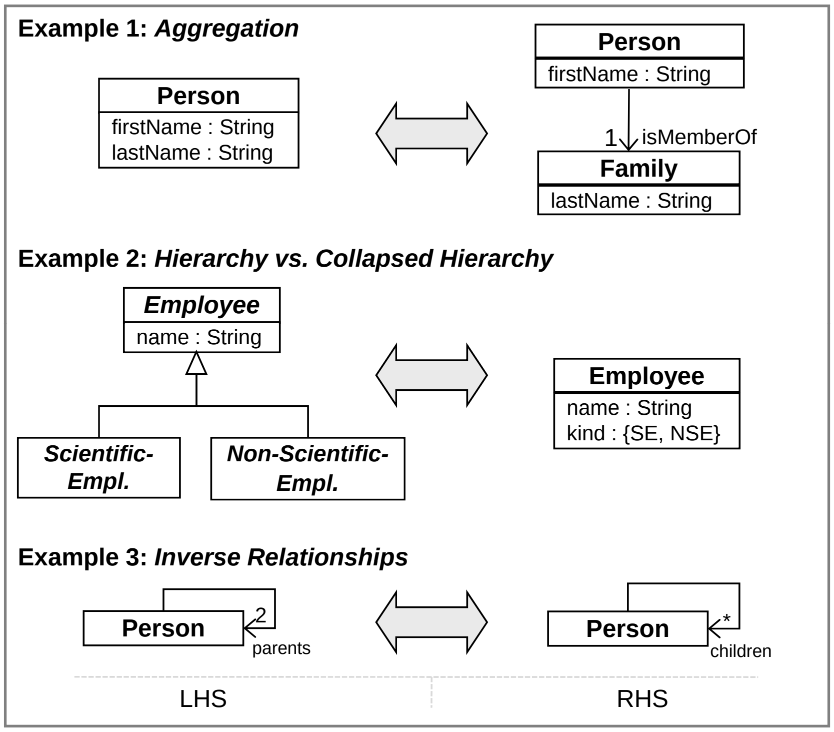


Fig. 1. Heterogenities in ontologies which cannot be resolved with simple mappings

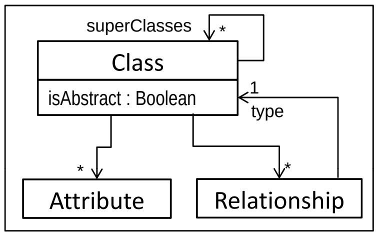
In all of the three examples, it is impossible to integrate the ontology parts on the left-hand side with the ontology excerpts on the right-hand side by simply applying equivalence mappings because classes are added, classes are eliminated, and relations are inverted. So more powerful operators are necessary which we present in the following.

3 Overview of CAR Mapping Language

The CAR mapping language is based on the combination of the core concepts of structural object-oriented modeling languages, i.e., *classes*, *attributes*, and *relationships*. The combination of these three concepts results in a 3x3 matrix of mapping operations (MOP) which is shown in Figure 2. These MOPs can be classified in three distinct categories. The first one is called *Copier* and consists of MOPs which copy elements from the left-hand side (LHS) into elements of the same kind at the right-hand side (RHS). This category comprises class-to-class (*C2C*), attribute-to-attribute (*A2A*), and relationship-to-relationship

(*R2R*) mapping operators. The second category is called *ObjectCreator*, which consists of operators for creating new objects on the RHS out of attribute values and links at the LHS. This category comprises two operators, namely attribute-to-class (*A2C*) for creating objects out of values and relationship-to-class (*R2C*) for creating objects out of links. The last category is called *Relationer*. MOPs from this category are *R2A*, *C2A*, *C2R*, and *A2R*. They have in common the creation of links between objects (C2R and A2R) or for defining which attribute value is contained by which object (C2A and R2A).

These basic MOPs can be combined to more complex patterns for solving more tricky problems. Complex patterns which are build from MOPs are called compound MOPs (CMOP) in the remainder of the paper. In the following, we first discuss the main properties of MOPs and afterwards we explain some CMOPs which provides mapping solutions to frequently occurring mapping problems.



	Class	*Attribute*	*Relationship*
Class	🟥 C2C	C2A	C2R
Attribute	🟦 A2C	🟥 A2A	A2R
Relationship	🟦 R2C	R2A	🟥 R2R

Legend

🟥 ... *Copier*
🟦 ... *Object Creator*
⬜ ... *Relationer*

Fig. 2. Possible combinations of modeling constructs

Copier Category. Operators from this category copies elements from the LHS to the the RHS. These operators can be used in both directions without any additional information and are therefore bidirectional.

- *C2C*: The C2C MOP can be used to map one class on the LHS to another class on the RHS (cf. Figure 3a). The interpretation of this mapping operator is, that the object identity is copied from the LHS to the RHS. C2C MOPs can inherit form each other which means that the sub-mappings inherit the contained property mappings of super-mappings. Furthermore, we allow the definition of abstract C2C mappings, typically for defining mappings between abstract classes.
- *A2A*: The A2A MOP copies attribute values of the LHS to attribute values on the RHS (cf. Figure 3b). Attributes are always in the context of a class,

hence an context relationship to an additional C2C MOP is needed to find the right context for the attribute at the RHS.
- *R2R*: The R2R MOP copies one link from the LHS to a link on the RHS (cf. Figure 3c). A relationship is contained in one object and refers to another one, hence R2R MOPs need two classes at the RHS. To derive these two classes, two additional C2C MOPs are required to set the two ends of the relationship to the proper classes at the RHS. To express these requirements, the R2R is in the context of two C2Cs. One context points to the C2C MOP where the reference belongs to. The other one belongs to the C2C where the reference points to.

Object Creator Category. The operators of the second category create new objects at the RHS from attributes or references on the LHS. These MOPs without any additional information can only be treated as one way operations. We only discuss the direction from left to right in Figure 3d and 3e. To make these operations bidirectional, additional information must be provided. The inversion of A2C and R2C, namely C2A and C2R representing *Relationer* MOPs are described in the next category description. Furthermore, the next category description also explains how the A2C and R2C are extended to bidirectional mapping operators.

- *A2C*: The A2C MOP creates a new object on the RHS for an attribute value on the LHS. An example application of A2C is shown in Figure 3d, where the attribute A1 is mapped to the class C2.
- *R2C*: The R2C MOP creates a new object on the RHS for a link on the LHS. Figure 3e illustrates such a mapping, namely relationship R1 on the LHS is mapped to class C4 on the RHS.

Relationer Category. The operators of the third category relates attribute values to objects and links objects based on the defined relationships correctly. Furthermore, we discuss how the mapping operators explained in the object creator category can be extended to bidirectional operators by proposing combinations of MOPs to CMOPs. In the following, we present C2A and C2R explicitly (we focus on the mapping from right to left in Figure3d and Figure3e); R2A and A2R are explained in the context of the C2A, because they only make sense in combination with C2A.

- *C2A*: (1) The C2A mapping in Figure 3d has no corresponding attribute value for the attribute A1 on the LHS, hence, the C2A needs an A2A to transfer the attribute value from the RHS to the LHS. (2) To find the right context of the attribute value on the LHS, the A2A mapping is in the context of the A2C mapping. (3.1) To derive the trace from C2A to C1 at the LHS, an additional C2C MOP is needed. (3.2)The R2A mapping is used to create a link between the class which is created with the A2C (C2 at the RHS) and the class where the attribute on the LHS belongs to and which is transfered with the C2C (C1 at the LHS to C1 on the RHS). The A2R needs the C2C and the A2C as a context for its two references to the classes on the RHS.

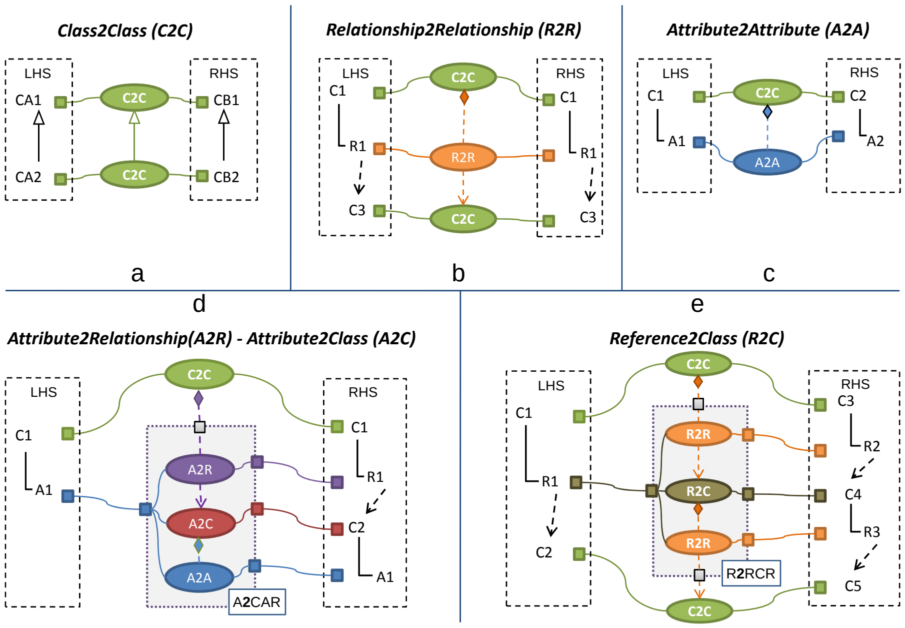


Fig. 3. CAR mapping operations

With this pattern of MOPs, the A2C from the ObjectCreator category is extended to a CMOP resulting in a bidirectional operator. The operator takes as input the attribute from the LHS and uses the A2R, A2A, A2C MOPs to create three outputs on the RHS, hence the resulting CMOP is called A2CAR. As mentioned before, A2R and R2A are only helper MOPs to make A2C and R2C MOPs bidirectional.

– *C2R*: (1) The C2R mapping operator is similar to C2A. To create a relationship (cf. relationship R1 in Figure 3e) out of a class at the RHS (cf. class C4 in Figure 3e), two classes on the LHS, namely C1 and C2 have to be mapped to the RHS with C2C mappings. (2) Now the class C4 on the RHS needs two references to the classes C3 and C5, which are transformed with the two C2C MOPs. This is done with two R2R MOPs. Each of them has two contexts, one to the class the reference belongs to and one to the class the reference points to.

The resulting mapping pattern gets a reference as an input from the LHS and uses the R2R, R2C, and R2R MOPs to create the outputs on the RHS, hence, the bidirectional version of R2C is a CMOP and called R2RCR.

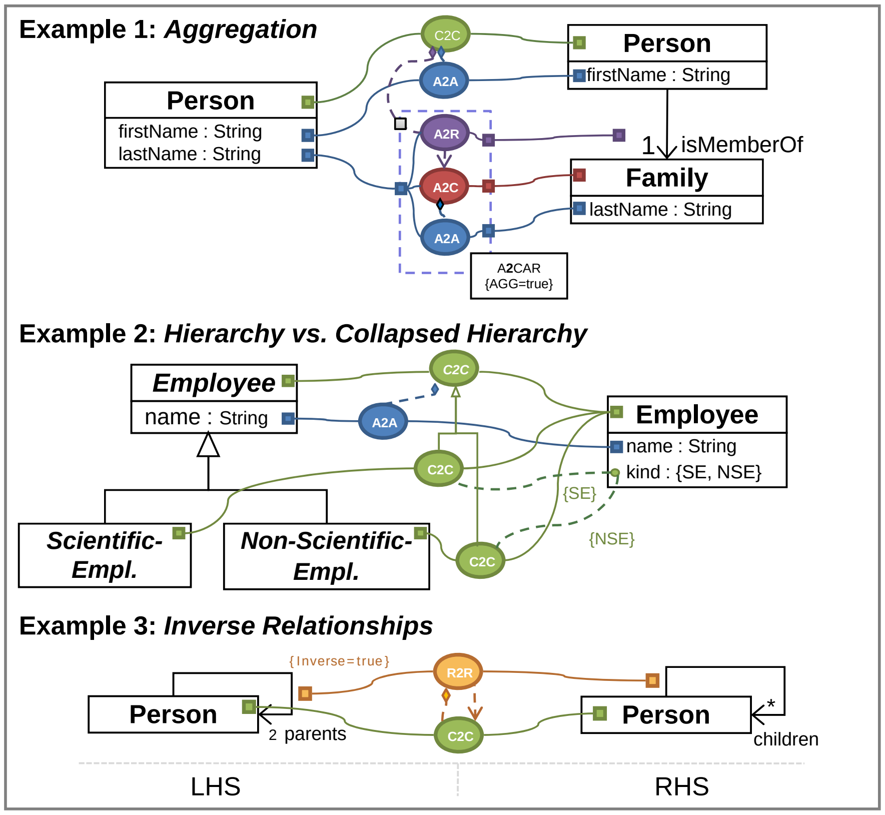


Fig. 4. By CAR-mappings resolved heterogenities

4 Application of CAR Mapping Language

With the previously introduced MOPs we are now able to resolve the heterogenities of the examples shown in Figure 1 and to integrate the ontology parts as follows (cf. Figure 4).

Example 1. First of all, we apply two BMOPs from the *Copier* category, namely we map *Person* to *Person* by C2C and we map the attribute *firstName* to the attribute *firstName* by A2A in Example 1. For the mapping of the *lastName*, we need additionally to the A2A the ObjectCreator operator A2C and the A2R operator from the Relationer category. The combination of those three BMOPs results in the CMOP A2CAR. Because of this structure of this mapping, we obtain bidirectionality. Note that it is necessary to aggregate on the last name when the mapping is implemented in order to avoid families with identical names.

Example 2. In Example 2 we flatten the inheritance hierarchy by mapping the subclasses *ScientificEmployee* and *Non-ScientificEmployee* of the abstract class *Employee* on the LHS to the attribute *kind* of the now concrete class *Employee*. Each concrete subclass of *Employee* at the LHS is mapped with a C2C to the concrete *Employee* on the RHS. The different kind of Employees on the LHS are represented as an attribute on the RHS, hence, the C2C has to set the attribute to the correct value of the enumeration datatype. To map the attribute name of the abstract class *Employee* on the LHS an additional abstract C2C between the abstract and the concrete class *Employee* is necessary. The concrete C2C operators between the concrete classes needs all the properties of the abstract BMOP, so they inherit from the abstract C2C.

Example 3. In the third example the main problem is to map two relationships which are inverse. This can be achieved by a R2R MOP with the inverse flag set to true. As described in the previous section a R2R needs a context to a C2C where it belongs to and one where it points to, hence, an additional C2C between the classes *Person* on the LHS and RHS is needed.

5 Mapping Execution and Tool Support

The first part of this section provides a description on how the declarative mappings are actually executed, which means how the instances are transformed between source and target schemas. Subsequently, the second part of this section describes the tool support for the mapping and transformation process. In particular, we present our mapping framework CARMEN (CAR Mapping ENvironment) and our transformation framework TROPIC (TRansformation On Petri nets In Color).

5.1 How to execute the mappings

For transforming instances between source and target schemas, executable transformation definitions must be provided. Therefore, we propose a transformation of CAR mapping models into a special kind of *Colored Petri Nets* [2] called *Transformation Nets*. In order to provide a mechanism to derive Transformation Nets from CAR mapping models, each CAR mapping operator has a corresponding Transformation Net component. In addition, this component defines the actual operational semantics of the CAR mapping operator.

In the following, we briefly present how instances, schemas and CAR mapping models are transformed into an executable transformation net. The interested reader is referred to [6] for detailed information on the notion of transformation nets and how they are actually executed.

– *Schemas*: Source schema elements are mapped into initial places and target schema elements into final places of the transformation net. A place is created for every class, attribute and reference of the schemas.
– *CAR mapping models*: Mappings are transformed into places and transitions between initial and final places being capable of reading tokens from the initial places and populate tokens in the final places.

– *Instances*: Instances are transformed into a certain marking of the initial places of transformation nets. For every model element that is an instance of a certain class, a one colored token is added to the place that corresponds to that class. The color of that token represents a unique object id. For every value of an attribute, a two colored token is added to the place which corresponds to that attribute. The two colors correspond to the object id of the owning element as well as the denoted value itself. For every link of a reference, a two colored token in the place corresponding to that reference is created. The colors correspond to the object ids of the linked objects.

The UML activity diagram in Figure 5 gives an overview of the complete process, which consists of two sub-processes, namely, mapping and transformation. In particular, the UML activity diagram depicts the tasks, input and output artifacts, as well as tool support involved in the two sub-processes. In the following subsection we discuss first our mapping framework CARMEN and afterwards our transformation framework TROPIC.

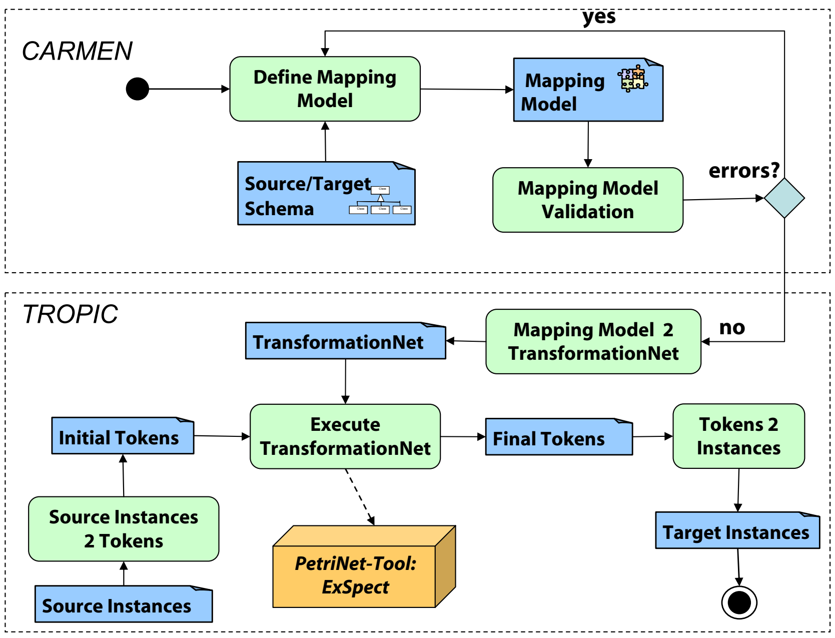


Fig. 5. Overview of the integration process

5.2 Tool Support

Mapping Framework CARMEN[3]

For providing a graphical mapping environment we developed a prototypical implementation of an GMF[4]-based editor, which is able to load two different schemas in UML class diagram notation. When the schemas are loaded, the user can build a graphical mapping model in between. The mapping model consists of the aforementioned CAR mapping operators. In addition, the mapping editor provides the validation of the mapping model that facilitates finding errors in the mappings. This possibility is applicable for situations in that mapping models are subsequently automatically processed as is the case in our approach.

Transformation Framework TROPIC[5]

After establishing a CAR mapping model, the actual transformations must be derived from these mappings. This task can be achieved manually, however, we believe that automation of this task is possible and preferable due to the formality of the mapping model. Therefore, we propose a full automatic transformation approach, so to speak a compilation of CAR mapping models into Transformation Nets. This compilation is achieved by a model transformation, which takes source and target schemas as well as CAR mapping models as input and produces textual petri net definitions readable for the Petri Net tool ExSpect[6]. In addition, we provide a second model transformation which is capable of producing initial tokens from an input model and a third model transformation which translates the final tokens (after executing the transformation nets) into a target model.

6 Conclusion and Future Work

Equivalent concepts can be expressed in different syntactical ways in general, and specifically when object-oriented modelling languages are used. These different modelling styles hamper the integration of ontologies considerable. In particular, practical integration issues demonstrate that *simple equivalence mappings* are not enough to overcome most of existing heterogeneities. Thus, in this paper we extended the notion of simple equivalence mapping with CAR mapping operators for resolving mapping problems due to different modelling styles, i.e., if a concept is modeled as a class, attribute, or relationship. Furthermore, we briefly presented the execution model for our CAR mapping language and proper tool support.

Concerning future work, we strive for a collection and categorization of schema heterogeneities and the evaluation of existing mapping and transformation languages.

[3] CAR Mapping ENvironment
[4] http://www.eclipse.org/gmf
[5] TRansformation On Petri-nets In Color
[6] http://www.exspect.com

Further research directions are finding on the one hand more compound mapping operators and on the other dedicated reasoning algorithms for a set of mappings. With reasoning algorithms we mean mechanisms for resolving possible problem cases in the transformation nets, which are automatically derived from mapping models. Figure 6 illustrates a mapping scenario which needs reasoning on the source and target schemas in combination with a set of mappings to correctly derive the transformation nets. In the LHS schema class A has a relationship to class B. In the right schema class A has a relationship to the abstract class X. A concrete subclass of X is for example class B. It is assumed that class A and class B of the RHS schema corresponds to class A and class B of the LHS schema, respectively. In addition, the user maps the relationship b to the relationship x. From these mappings a correct transformation net for transforming instances from the LHS to the RHS is derivable without additional processing steps. However, then going the other direction, namely from RHS to LHS, the relationship x can contain instances of class C and D in addition which have no correspondences in the LHS schema. Therefore, in the transformation process these instances must be deleted out of the relationship set, because relationship b can contain instances of class B, only. Providing such reasoning algorithms in our mapping framework to resolve possible problem cases is subject to ongoing work.

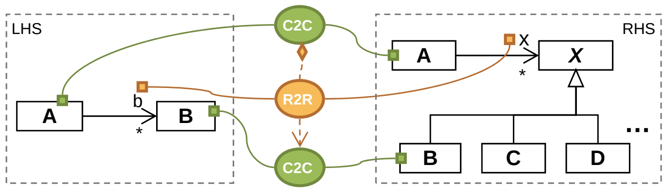


Fig. 6. Mapping scenario requiring additional reasoning

References

1. M. Fowler. *Refactorings.* Addison-Wesley, 2003.
2. K. Jensen. Coloured petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer, 1986.
3. F. Jouault and I. Kurtev. Transforming models with ATL. In J.-M. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844*, pages 128–138. Springer Berlin / Heidelberg, 2006.
4. V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: a context-based approach. *The VLDB Journal*, 5(4):276–304, 1996.

5. F. Legler and F. Naumann. A classification of schema mappings and analysis of mapping tools. In *Proceedings of the BTW*, volume 103 of *LNI*, pages 449–464. GI, 2007.
6. T. Reiter, M. Wimmer, and H. Kargl. Tropic - transformation on petri-nets in color. *Submitted for publication*, 2007.
7. A. P. Sheth and V. Kashyap. So far (schematically) yet so near (semantically). In *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, pages 283–312. North-Holland Publishing Co., 1993.
8. S. Spaccapietra and C. Parent. Conflicts and correspondence assertions in interoperable databases. *SIGMOD Rec.*, 20(4):49–54, 1991.