CARE – A Constraint-Based Approach for Re-Establishing Conformance-Relationships

Johannes Schoenboeck[1] Angelika Kusel[2] Juergen Etzlstorfer[2]
Elisabeth Kapsammer[2] Wieland Schwinger[2] Manuel Wimmer[3]
Martin Wischenbart[3]

[1] University of Applied Sciences Upper Austria, Campus Hagenberg, Austria
[firstname].[lastname]@fh-hagenberg.at

[2] Johannes Kepler University Linz, Austria
[firstname].[lastname]@jku.at

[3] Vienna University of Technology, Austria
[lastname]@big.tuwien.ac.at

Abstract

In Model-Driven Engineering, models have to conform to their associated *linguistic* and *ontological* metamodels. While linguistic metamodels are usually not subject to frequent changes, ontological metamodels are. Thus, existing conformance relationships may be easily corrupted by changes on the metamodel or by the incautious modification of models. Current approaches for re-establishing conformance relationships are often (i) deeply woven into specific tools to record changes and to derive resolutions, or (ii) require extensive user effort to guide the resolution process, and (iii) the output of these approaches usually is one single solution, whereas alternative solutions remain unexplored. To allow for *exploring a broader solution space* independent of specific tools and to *avoid extensive user involvement* by utilizing predefined repair actions, we propose a logic programming approach called CARE, for accomplishing *multiple solutions.* In particular, CARE bases on a *formalization* of the *ontological conformance relationship* as constraints, accompanying *repair actions* for counteracting constraint violations, as well as *quality criteria* for ranking of solutions. This paper reports on the realization of CARE based on Answer Set Programming and summarizes lessons learned from applying the approach in several experiments.

1 Introduction

Model-Driven Engineering (MDE) [26] proposes a continuous use of models to conduct the different phases of software development. Models have to conform to their associated linguistic and ontological metamodels that are prevalent in a certain domain and which define concepts, their relationships, as well as constraints among each other. Consequently, conformance between models and their associated metamodels can be classified into *ontological conformance*, based on the *meaning* (e. g., an object Mickey is an *instance of* a class Mouse), and *linguistic conformance*, regarding their *syntactical form* (e. g., Mickey is an *instance of* Object) [16].

This work has been funded by BMVIT under grants FFG BRIDGE 832160 and FFG FIT-IT 825070 and 829598, FFG Basisprogramm 838181, and by ÖAD under grant AR18/2013 and UA07/2013.

While linguistic metamodels, such as Ecore[1], are often standardized and changed seldomly, ontological metamodels, representing concepts within a certain domain, are frequently subject to change [12]. This refers to the typical case of metamodel evolution, which entails the co-evolution of dependent artifacts in order to retain conformance [5, 7, 28]. Further, in data integration scenarios, conformance of existing models to a new metamodel may be disrupted [2], and, thus, has to be re-established. Moreover, the incautious modifications of models may also violate ontological conformance, while keeping the models syntactically correct, thereby maintaining linguistic conformance. In either case, model processing is obstructed in current tools, until conformance between models and their ontological metamodels is re-established, which is, therefore, the focus of this paper.

Since the manual re-establishment of conformance is tedious and error-prone, dedicated (semi-) automatic approaches exist, which are, however, (i) often *tightly coupled* to specific tools to record changes and to derive repair actions, or (ii) require *extensive user effort* in the resolution process, e. g., by demanding to specify the resolution steps beforehand, or by guiding the tool step-by-step. If changes cannot be tracked or derived due to the absence of the prior metamodel version, it is of special interest to automatically re-establish conformance without relying on manually provided input. Finally, representatives of both kinds of approaches mostly (iii) provide a single solution, only, *disregarding alternative solutions* with respect to their qualitative properties.

To overcome these shortcomings, we propose a framework for *Constraint-bAsed REpairing of ontological conformance relationships* (CARE), which is (i) *independent of a specific tool* by means of a stand-alone framework instead of being deeply woven into an existing modeling environment, (ii) utilizes *predefined repair actions*, and, thus, requires *little user involvement* for specification as well as guidance of the resolution process, and is (iii) capable of generating *multiple, ranked solutions*, which fit a set of *quality criteria* that should be naturally fulfilled by repair solutions, such as the preservation of information capacity.

Operating in *three phases*, CARE is able to re-establish ontological conformance, independent

[1]Ecore is the realization of MOF [22] in the Eclipse Modeling Framework (EMF) `http://www.eclipse.org/modeling/emf`

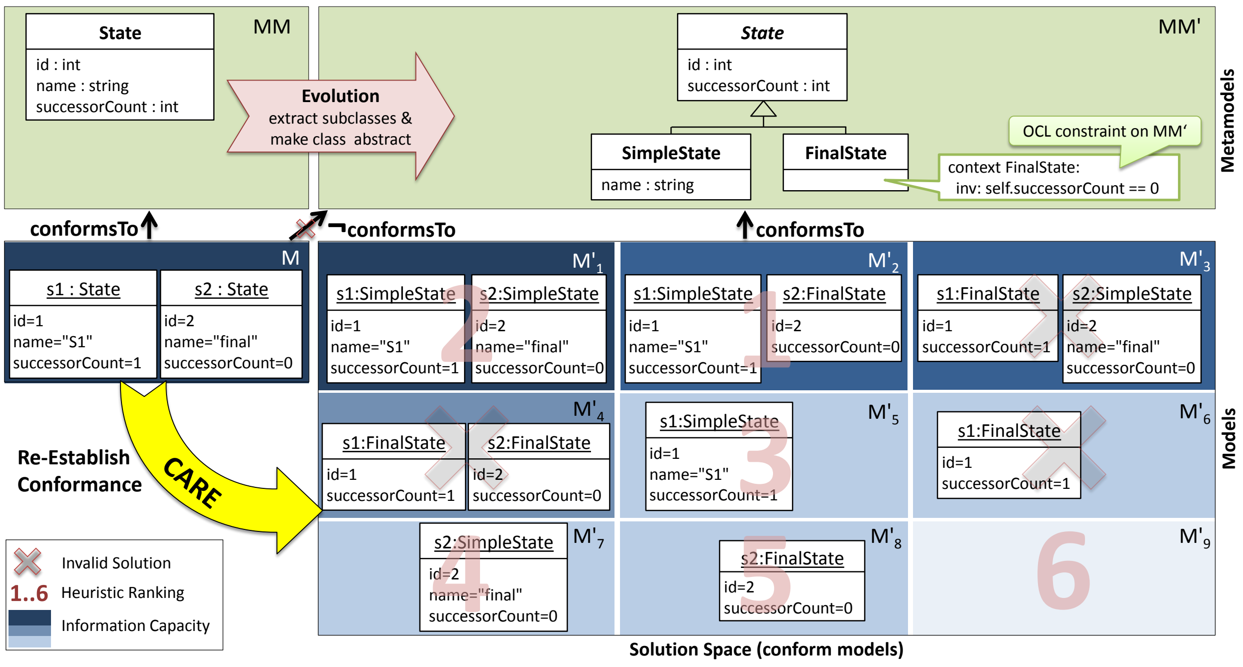


Figure 1: Example for Conformance Re-Establishment: Retyping & Deletion of Objects

whether changes in the model or in the metamodel caused the violation. Therefore, CARE bases on the core concepts of Ecore, having classes, attributes, and references with respective objects, values, and links. In phase ① *constraint violations* are *detected* based on a formalization of the ontological conformance relationship in form of dedicated constraints in logic programming, in phase ② *constraint violations* are *repaired* by means of repair actions, generating multiple ontological conforming solutions, and in phase ③ *ranking* and *selection* of best solutions is achieved by applying quality criteria, which comprise structural and semantic knowledge about the potential solution space. Furthermore, *provenance information* and statistics are collected during all phases and reported to the user for inspecting the solutions.

Before the three phases of the CARE approach are presented in detail in Section 3, Section 2 introduces a motivating example. Section 4 demonstrates a proof-of-concept prototype based on Answer Set Programming, while lessons learned from several experiments are presented in Section 5. Finally, related work is discussed in Section 6 before Section 7 concludes.

2 Motivation

This section introduces a motivating example and thereupon outlines different ways to deal with the challenges to *automatically* re-establish *ontological conformance* based on different kinds of knowledge. Figure 1 shows a simple example, basing on UML state machines, with a model M comprising two State objects, whose conformance gets violated due to metamodel evolution comprising three steps: *extract subclass* for State resulting in SimpleState and FinalState as well as *make class abstract* for State[2]. These steps, however, do not need to be known for re-establishing conformance of M to the evolved metamodel MM', since CARE is capable of re-establishing conformance regardless whether changes in the metamodel or in the model caused the violation.

[2]Note, that classification of evolution steps bases on [11] and [14].

Solution Space. To re-establish conformance in the example, non-conforming objects (cf. s1 and s2 in Fig. 1) may either be *retyped* (reclassified as instances of the concrete classes SimpleState or FinalState) or *deleted*. By applying these options, a total of nine solutions with respect to model-metamodel constraints arises, so far disregarding any constraints on the metamodel. Thus, the potential solution space for retyping or deleting non-conforming elements contains $(c+1)^o$ solutions (with c = number of candidate classes + 1 for deletion, o = number of non-conforming objects).

Selection. These solutions, however, need not to be valid, since metamodel-specific constraints may be violated. Consequently, for selecting valid (i. e., conforming) solutions from the solution space, metamodel-specific constraints, e. g., specified in the Object Constraint Language (OCL), and user-provided constraints, such as mapping instructions, must be validated. Thus, given the OCL constraint in Figure 1, the solutions M'_3, M'_4, and M'_6 are invalidated.

Ranking. The remaining set of valid solutions may further be ranked, to provide the user with ordered solutions pushing forward those solutions fulfilling dedicated qualitative criteria. For this, heuristics incorporating structural and semantic knowledge may be employed. Regarding structural knowledge, the structural similarity between the objects in M and M' as well as between M and MM' may be employed, by using methods which are similar to existing schema and ontology matching techniques [6, 25]. In contrast, matching has to be done across metamodeling layers. Furthermore, in the context of ranking, the prevention of information loss (indicated by shades in Fig. 1) may be considered essential. Exploiting this structural knowledge, M'_1 scores best, preserving all attributes, and M'_9 is ranked last, since all model elements are deleted. Additionally, semantic knowledge obtained from an external knowledge base (e. g., an ontology) or provided by the user in terms of domain knowledge, may be used for ranking. Given the knowledge that the value “final” of attribute label rather corresponds to FinalState than to

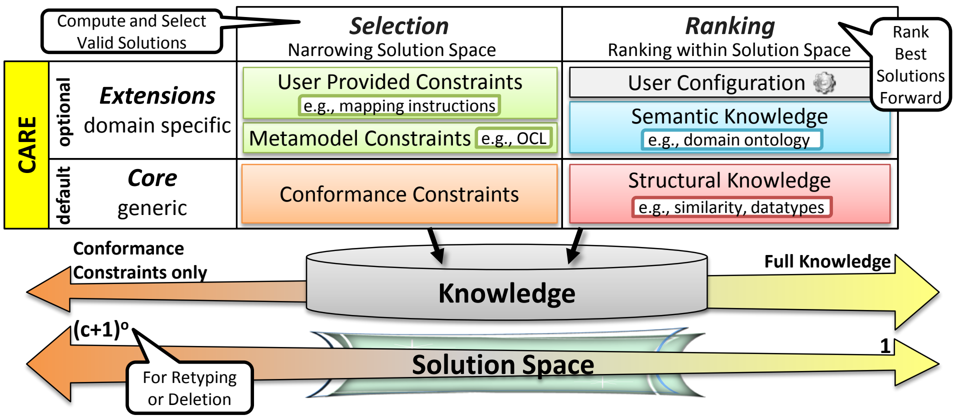


Figure 2: Classification of Available Knowledge in Correlation to Solution Space

SimpleState, retyping of s2 to FinalState is preferred, thus, favoring M'_2 over M'_1, as indicated in Figure 1. Consequently, the ranking of valid solutions depends on a particular configuration, which may favor different quality criteria. Although CARE provides a default configuration, e. g., trying to prevent information loss, users may overrule it by providing a custom configuration.

To summarize, Figure 2 gives a classification of the sources of knowledge, comprising constraints for *selection* of valid solutions, and heuristics to be utilized for *ranking* of solutions. Given these sources of knowledge, the CARE framework may be divided into a *generic core* part, comprising *conformance constraints* and *structural knowledge*, and an optional *extensions* part, which is domain specific. The extensions part may be divided into *constraints*, comprising *metamodel-specific* ones (e. g., specified in OCL) and *user-provided* ones (e. g., mapping instructions), and *semantic knowledge* provided by users or external knowledge bases, as well as the user configuration. Consequently, although the CARE framework may provide multiple valid solutions by relying on the generic *core* part, only, the more information is available for the *extensions* part, the better the *solution space* may be *narrowed*, as indicated in Figure 2. To achieve the goal of re-establishing conformance, CARE operates in three phases, which are described in detail in the next section, together with an overview of the approach.

3 CARE Process: Detection, Repairing, Ranking

The main idea of CARE is to re-establish conformance between non-conforming models and their metamodels by utilizing ontological conformance relationships, formalized as constraints between a model and its metamodel. Based on corresponding repair actions, CARE generates multiple ranked solutions facilitated by Answer Set Programming (ASP) [17]. To achieve this goal, CARE operates in three phases as illustrated in Figure 3.

As a prerequisite, the non-conforming model, as well as the metamodel together with its constraints are transformed into ASP. For that, CARE bases on the core concepts of metamodeling, consisting of classes, attributes, and references, i. e., a representative subset of Ecore focusing on structure, not considering operations[3]. Ecore, however, may be replaced by other modeling languages, by implementing the appropriate transformations and by modifying the conformance constraints, repair actions and ranking rules.

In the first phase, *violations are detected* (cf. ① in Fig. 3) by means of ASP rules which employ built-in conformance constraints (e. g., objects have to be instances of non-abstract classes), as well as optional metamodel-provided and user-provided constraints. Second, during *model repairing* (cf. ② in Fig. 3), all possible solutions that re-establish ontological conformance are calculated by means of corresponding repair rules. In order to *rank* the resulting *models* in a third phase (cf. ③ in Fig. 3), built-in heuristics (e. g., based on structural similarity) together with external semantic knowledge (e. g., for semantic matching) may be employed in so-called ranking rules. This means that for repairing and ranking a "guess, check & optimize" methodology [9] is applied, which allows to discard non-optimal solutions early during their computation, and thereby allows to speed up the whole process. Since all three phases operate on the same knowledge base of logic programming facts and rules, CARE enriches solutions with meta information by using intermediate results to generate provenance information and statistics, which, ultimately, support the user in inspecting differences between solutions after the fulfilled transformation back to Ecore.

3.1 Phase 1: Detection of Violations

For the detection of conformance violations between a model and its metamodel, CARE bases on formalized constraints, i. e., checking conditions for conformance on all model elements. In general, conformance is accomplished, if each model element can be syntactically regarded as a valid instance of a type in the metamodel [16]. Applied to the core concepts of Ecore, this means that model elements, namely *objects* with features (*values* and *links*), have to conform to respective *classes* with class features (*attributes* and *references*) in a metamodel. Therefrom, we derived a more specific definition, formulated as a set of object, value, and link constraints, as listed in Table 1. Thus, object constraints are denoted with ocX, value constraints with vcX and link constraints with lcX, respectively, whereby the X represents an ID. For instance, for the type of an object a corresponding class must exist (cf. oc1), which must not be abstract (cf. oc2, depending on oc1). Besides such generic constraints provided by the CARE core, constraints on the metamodel in terms of OCL, as well as user-

[3]Packages, operations, data types, enumerations, as well as opposite and composite references are currently not considered.

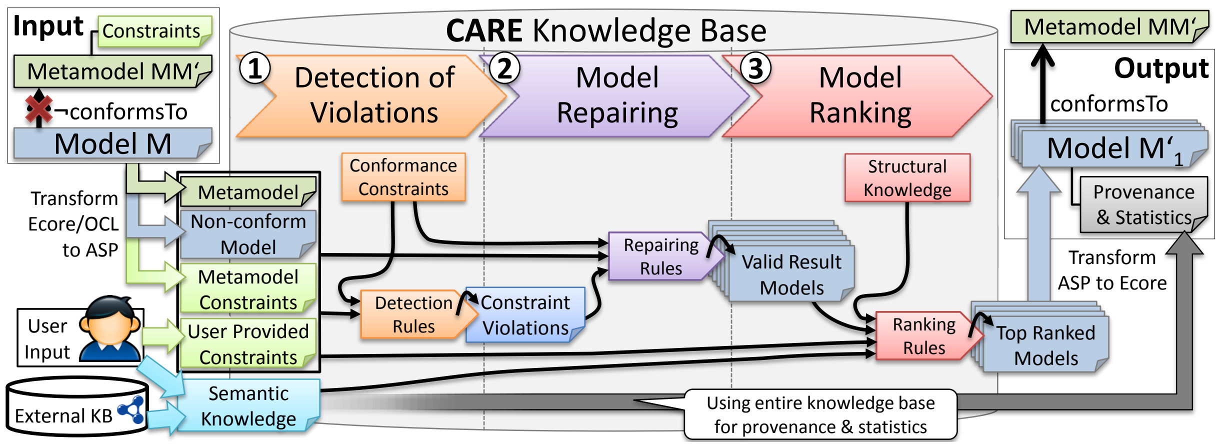


Figure 3: CARE Process Overview: Input, Knowledge Base with 3 Phases, Output

provided ones in terms of mapping instructions may be considered to validate conformance of model elements (cf. CARE extensions). An example for an OCL constraint was shown in the introductory state machine example (cf. Fig. 1), defining that a FinalState must not have successors, i. e., successorCount must be 0.

To check whether all conformance constraints are fulfilled, they are formulated as separate interdependent logic programming rules. A constraint solver evaluates these rules to produce the output, i. e., facts describing which objects, values, and links are conform or not, and adds these new facts to the CARE knowledge base. Examples for such evaluations are shown as implication rules in Table 2, using a pseudocode language in the style of ASP, with conjunction denoted by $\wedge$ (cf. Fig. 4 in Sect. 4 for corresponding ASP code). The above mentioned object constraint oc2 is expressed in the form of two rules, shown in lines 3-4, stating that an object with a corresponding, non-abstract class fulfills oc2, whereas an object not fulfilling the rules for oc2 (which also requires oc1) violates the constraint (cf. `oc2violated(OID)`). In line 9, the above mentioned OCL constraint is specified formally, which is currently achieved manually by the CARE user, but foreseen to be automated and subject to future work. Note, that `object(OID,CID)` denotes that an object with ID `OID` is typed to a class with ID `CID`.

Based on these detection rules, for the introductory example the resulting violations are `oc2violated(s1)` and `oc2violated(s2)`, since both State s1 and State s2 are typed to an abstract class in MM'. These detected violations are then used as inputs for repair actions in the next phase.

3.2 Phase 2: Model Repairing

Based on the violations from the first phase, another set of rules allows to re-establish ontological conformance between a model and its metamodel. For each kind of violation, CARE uses dedicated repair actions, which are systematically derived from the constraint violations listed in Table 1 and, thus, predefined within the CARE framework. Basically, the options to repair violations are to *modify* or to *delete* the violating model element. For instance, an object, which violates oc2 can either be retyped to another class (i. e., the type reference from the object to its class is modified, cf. ra1), or the object can be deleted (cf. ra2). Such alternative repair actions are denoted as rows in Table 1. Table 2 shows the examples discussed above in pseudocode. A disjunction for rules is expressed by using $\vee$, thereby producing all possible different solutions, i. e., retyping and deletion. Line 5 in Table 2 states, that if `oc2` is violated, either repair action `oc2ra1` (retype object) or `oc2ra2` (delete object) has to be applied. For producing output elements, additional facts must be added to the knowledge base, which are denoted with suffix `x`. For the deletion of elements, no further action is required in `oc2ra2` (i. e., no `objectx` is added), since elements with suffix `x` are regarded as output elements, only. Consequently, the suffix `x` is also applied to all valid elements to declare them as output elements. Note, that on the left hand side of the rule in line 6 of Table 2 there is no binding of the object's ID to a specific class ID. Thus, the object is retyped to all possible non-abstract classes, and consequently, all possible combinations of retyping are generated. However, since an object can only be the instance of one class at a time, another rule must prevent these solutions from being generated (cf. `PREVENT` rule in line 7 of Table 2).

For the example, this means that the two objects which violate oc2 (s1 and s2), can either be deleted, retyped as SimpleState, or retyped as FinalState. By applying additional metamodel constraints by means of OCL constraints (cf. line 10 in Table 2), invalid solutions are excluded (M'_3, M'_4, and M'_6). Finally, concerning the example, only the remaining six valid solutions need to be ranked in the next phase, to ultimately present a ranked list of models to the user.

3.3 Phase 3: Model Ranking

After extracting valid solutions that meet both, conformance constraints and metamodel-specific constraints, in the third phase, solutions are evaluated and *ranked* using configurable rules, based on heuristics, which incorporate structural as well as semantic knowledge. Together with accompanying meta information about solutions, this ranking enables the user to inspect the best provided solutions, in order to ultimately select one.

For finding the best solutions, CARE assigns costs to each solution, by using a third kind of rules, which, instead of preventing a solution, incurs a given cost to be added up. These rules, however, may also base on other rules for more complex computations, and they may incorporate specific facts about the domain. For that, as discussed in Section 2, CARE differs between structural and semantic knowledge. Rules based on structural knowledge, for instance, may use match-

	ID	Definition of Constraint/Repair Action	Req.
Object	oc1	For the type of an object a corresponding class must exist. Required Preconditions	-
	ra1	Change type of object [multiple solutions]	
	ra2	Delete object with values and links	
	oc2	Corresponding class must not be abstract.	oc1
	ra3	**ra1 ∨ ra2**	
Value	vc1	For all values of an object, a corresponding attribute in the corresponding class (or in its superclasses) must exist.	oc2
	ra4	Use another attribute from the class (or a superclass) [multiple solutions]	
	ra5	Delete value	
	ra6	**ra1 ∨ ra2**	
	vc2	For all (inherited) attributes in a class, a corresponding object must fulfill minimal cardinality of values.	-
	ra7	Add missing attributes using default values	
	ra8	**ra1 ∨ ra2**	
	vc3	For all (inherited) attributes in a class, a corresponding object must fulfill maximum cardinality of values.	-
	ra9	Delete values (randomly) [multiple solutions]	
	ra10	**ra1 ∨ ra2**	
Link	lc1	For all links of an object, a corresponding reference in its corresponding class (or in its superclasses) must exist.	oc2
	ra11	Use another reference with matching types from the corresponding class (or its superclasses)	
	ra12	Delete link	
	ra13	**ra1 ∨ ra2**	
	lc2	For all (inherited) references in a class, a corresponding object must fulfill minimal cardinality of links.	-
	ra14	Create link with target null	
	ra15	Create link with existing target object (randomly) which has the correct type [multiple solutions]	
	ra16	**ra1 ∨ ra2**	
	lc3	For all (inherited) references in a class, a corresponding object must fulfill maximum cardinality of links.	-
	ra17	Delete links (randomly) [multiple solutions]	
	ra18	**ra1 ∨ ra2**	
	lc4	For all links of an object, the target object's type must be the class defined by the reference (or its subclasses).	lc1
	ra19	Keep the link, but refer to another object with matching type	
	ra20	Keep the link, but set target to null	
	ra21	**ra1 ∨ ra2 ∨ ra11 ∨ ra12**	
	lc5	For all links refering to an object (i.e., incoming links), a corresponding reference in the refering object's class (or in its superclasses) must exist.	oc2
	ra22	**ra1 ∨ ra2**	

Table 1: List of CARE Core Conformance Constraints with Repair Actions

ing heuristics, or determine information capacity of solutions, based on inputs, outputs, and intermediate results, and, thereupon, assign costs for structural difference or information loss. Additionally, semantic knowledge from external knowledge bases, such as domain ontologies or linked data, or provided by users, may be used for such heuristics as well. Another interesting criterion in the course of CARE is whether a heuristic employs single model elements, whole objects including features, or the complete model, i. e., the granularity. For instance, computation of similarity between input and output models may base on matching of single objects, values, and links by ID, on the comparison of the number of features per class, or on the usage of characteristics of the complete model, such as connectivity between objects. A more detailed classification of this knowledge, however, would go beyond the scope of this paper. Consequently, different kinds of knowledge allow for a variable configuration of the ranking phase, firstly by adding or removing rules and facts, and secondly, by adjusting

		ID		Rule in Pseudocode
Core	1	oc1		`object(OID,CID) ∧ class(CID) ⇒ oc1(OID)`
	2			`object(OID,CID) ∧ NOT oc1(OID) ⇒ oc1violated(OID)`
	3	oc2		`object(OID,CID) ∧ oc1(OID) ∧ NOT isAbstract(CID) ⇒ oc2(OID)`
	4			`object(OID,CID) ∧ NOT oc2(OID) ⇒ oc2violated(OID)`
	5		ra1/2	`oc2violated(OID) ⇒ oc2ra1(OID) ∨ oc2ra2(OID)`
	6		ra1	`oc2ra1(OID) ∧ class(CID) ∧ NOT isAbstract(CID) ⇒ objectx(OID,CID)`
	7		ra1c	`PREVENT objectx(OID,CID1) ∧ objectx(OID,CID2) ∧ CID1!=CID2`
	8		rank1	`COST[2] oc2ra2(OID)`
Extensions	9	mm1		`value("succCount",VAL,OID) ∧ object(OID,"FinalState") ∧ VAL!=0 ⇒ mm1violated(OID)` OCL
	10		mm1c	`PREVENT valuex("succCount",VAL,OID) ∧ objectx(OID,"FinalState") ∧ VAL!=0`
	11	rank2		`COST[1] value("label","final",OID) ∧ NOT objectx(OID,"FinalState")`

Table 2: Pseudocode for Selected Conformance and Metamodel Constraints

cost parameters.

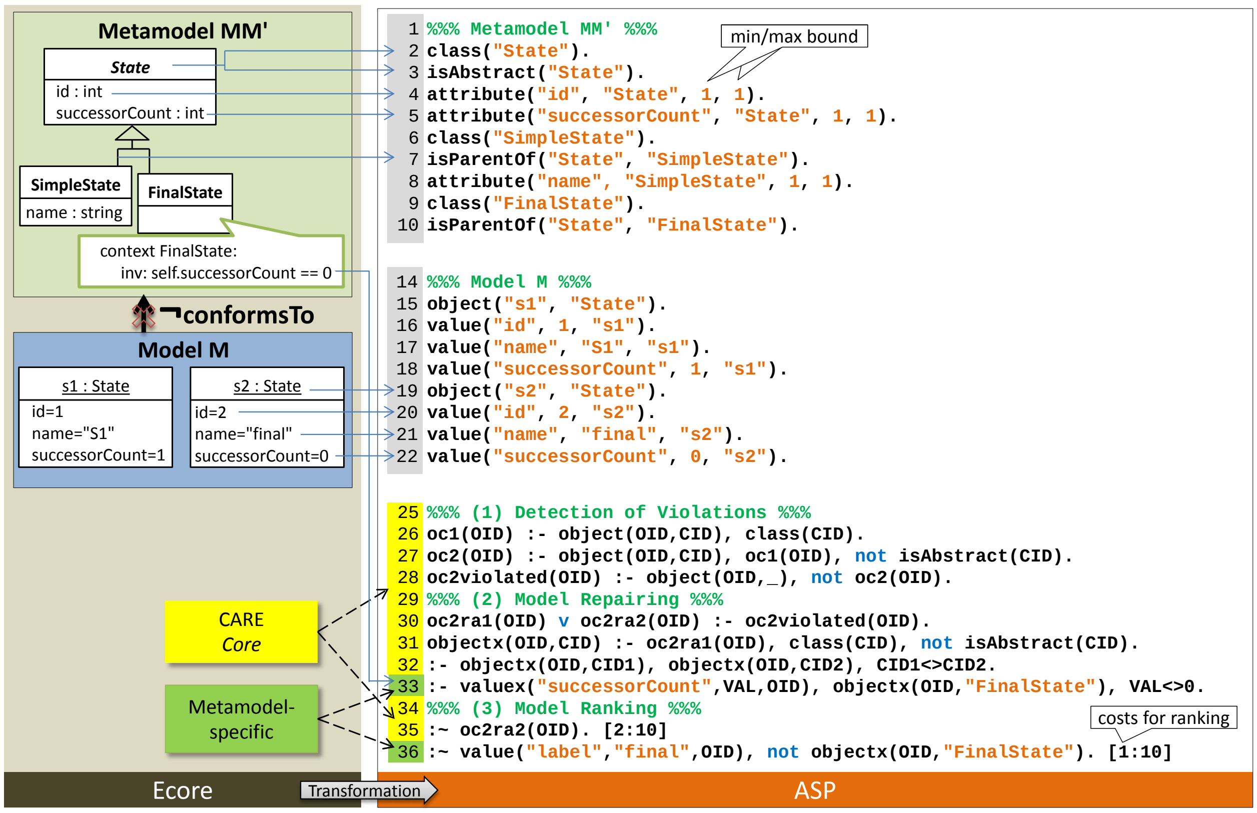
Again, Table 2 shows a simple example of such heuristics, as used in the introductory example. The rule in line 8 restrains information loss, by assigning costs to solutions where repair action oc2ra2 (delete object) is taken. As additionally specified, line 11 assigns costs for objects with label "final" that are not retyped to FinalState. Thus, for the example, this means that M'_2 is the best solution (cost = 0), and therefore ranked first. M'_1 is ranked second, since s2 with label "final" was retyped to SimpleState (cost = 1). Concerning the remaining four, M'_9 is ranked last, since both objects were deleted (cost = 4).

As a result, CARE provides a list of ranked solutions, which are transformed back to Ecore. To support the user in inspecting and choosing solutions, additional meta information in terms of provenance information and statistics is provided as annotations using the dedicated Open Provenance Model (OPM) [20] vocabulary. Thus, a model element in M' is an *artifact* in OPM, which is *derived from* another artifact in M, and which is *used* in processes, i. e., CARE repair rules. This information may be derived directly from intermediate results, whereas statistics, in contrast, require additional rules. For instance, the retyping of s1 in M'_2 is expressed as s1:SimpleState wasDerivedFrom s1:State. For counting how many states were retyped to SimpleState, however, an additional computation step is required, which may also be performed on the transformed Ecore model. Ultimately, this meta-information may be presented to the user to facilitate the inspection of the ranked solutions.

4 Proof-of-Concept Prototype

After having discussed the three phases of CARE, this section presents the prototype for re-establishing ontological conformance between Ecore-based models and metamodels, implemented with ASP.

The CARE proof-of-concept prototype supports the core concepts of Ecore, as discussed in the previous section, as well as OCL constraints. All these inputs may be represented as logical axioms in ASP, thereby resulting in a similar representation to those presented in [4, 24]. Consequently, transformations between Ecore/OCL and ASP are required, which are



```
 1 %%% Metamodel MM' %%%
 2 class("State").
 3 isAbstract("State").
 4 attribute("id", "State", 1, 1).
 5 attribute("successorCount", "State", 1, 1).
 6 class("SimpleState").
 7 isParentOf("State", "SimpleState").
 8 attribute("name", "SimpleState", 1, 1).
 9 class("FinalState").
10 isParentOf("State", "FinalState").

14 %%% Model M %%%
15 object("s1", "State").
16 value("id", 1, "s1").
17 value("name", "S1", "s1").
18 value("successorCount", 1, "s1").
19 object("s2", "State").
20 value("id", 2, "s2").
21 value("name", "final", "s2").
22 value("successorCount", 0, "s2").

25 %%% (1) Detection of Violations %%%
26 oc1(OID) :- object(OID,CID), class(CID).
27 oc2(OID) :- object(OID,CID), oc1(OID), not isAbstract(CID).
28 oc2violated(OID) :- object(OID,_), not oc2(OID).
29 %%% (2) Model Repairing %%%
30 oc2ra1(OID) v oc2ra2(OID) :- oc2violated(OID).
31 objectx(OID,CID) :- oc2ra1(OID), class(CID), not isAbstract(CID).
32 :- objectx(OID,CID1), objectx(OID,CID2), CID1<>CID2.
33 :- valuex("successorCount",VAL,OID), objectx(OID,"FinalState"), VAL<>0.
34 %%% (3) Model Ranking %%%
35 :~ oc2ra2(OID). [2:10]
36 :~ value("label","final",OID), not objectx(OID,"FinalState"). [1:10]
```

Figure 4: Ecore Representation and ASP Code for Input Model M, Metamodel MM', as well as Rules for Detection of Violations, Model Repairing and Model Ranking

already automated for Ecore using Xtend[4], and foreseen to be automated for OCL by utilizing methods presented in [19] and [24]. Rules and constraints for all three phases are implemented in ASP, which is a form of declarative programming language, based on the stable model semantics of logic programming [17]. An ASP program is a finite set of *rules*, consisting of a head (on the left) and a body part (on the right). In CARE, such rules are used for detecting and repairing violations and as basis for ranking. Along with two special kinds of rules, they constitute CARE's knowledge base. First, rules with empty body represent *facts*, describing models and metamodels [4]. Second, rules with empty head, called ASP *constraints*[5], are used to restrict repair actions and, thereby, prevent generation of invalid solutions. Figure 4 shows an excerpt of such an ASP program, including the transformed metamodel and input model from Figure 1 and the ASP representation of the rules from Table 2.

Encoding of Models and Metamodels. For the subset of Ecore specified in Section 3, each metamodel element is transformed into an according ASP fact, i. e., `class`, `attribute`, and `reference`. Regarding the motivational example, a class fact is defined for each class, i. e., `class('State')`, `class('SimpleState')`, and `class('FinalState')`. Additional constructs (attributes of classes, relationships between elements) have been systematically derived from the Ecore subset, and corresponding ASP facts were introduced. Thereby, facts for attributes also contain the cardinality, represented by integers, e. g., a fact `attribute('id', 'State', 1, 1)` defines the attribute `id` for the class `State` with a minimum and maximum cardinality of 1. Regarding the motivational example, an additional fact for the abstract class `State` is added, i. e., `isAbstract('State')`, as well as facts for the hierarchy relationships, i. e., `isParentOf('State','SimpleState')`, and `isParentOf('State','FinalState')`.

In a similar manner, the model elements object, value, and link are transformed to the ASP facts `object`, `value`, and `link`, respectively. Thereby, facts for objects consist of the name and type of the object, resulting in `object('s1','State')` and `object('s2','State')` for the example. Values are represented by their id, the actual value, and the object they belong to, resulting in, e. g., `value('id', 1, 's1')`.

Encoding of Constraints and Repair Actions. Based upon this representation, the conformance constraints, repair actions, and generic ranking rules representing the generic core part have been manually defined once (cf. rules 1-8 in Table 2). Figure 4 shows the ASP code required for re-establishing conformance for the motivational example. Lines 25–32 as well as line 35 are part of the CARE core functionality, while line 33 and line 36 have been derived from the metamodel-specific OCL expressions. OCL constraints consisting of *boolean* expressions, *select*, or *size* operations are candidates to be transformed automatically into ASP constraints (cf. rules 9-10 in Table 2), which may be established utilizing the approach of [24] in future versions of the prototype. In contrast, domain-specific ranking rules have to be specified manually (cf. rule 11 in Table 2, which has been transformed to line 35 in Figure 4). Although not required for generating valid solutions, they facilitate proper ranking of these solutions.

Re-Establishing Conformance. To execute

[4]http://www.eclipse.org/xtend/
[5]ASP constraints are dissimilar to the notion of constraints in this paper.

ASP programs, CARE employs the DLV[6] solver, which provides several extensions to ASP, such as the support for *weak constraints* [9], i. e., a special kind of ASP constraints to specify costs for non-compliance of constraints. Cost bounds and a limitation in terms of the number of results can be used to rank and restrict the resulting models, in order to compute best solutions meeting these quality criteria, only. In particular, computation can be speeded up, since solutions that score worse than the current cost bound can be eliminated early. The generated solutions are then transformed back into Ecore, allowing the user to select one of the ranked solutions.

5 Lessons Learned

For the experiments we extended the introductory example based on the UML 1.4 state machine metamodel, focusing on re-establishing ontological conformance of objects. In the following, we present lessons learned gained from these experiments.

CARE Core Depends on Structural Diversity. Given that CARE is used solely on basis of the generic core part, the ranking quality heavily depends on structural diversity of the classes in the metamodel. Consequently, if classes in the metamodel are structurally similar and no additional knowledge sources are incorporated, CARE suffers from the theoretically large solution space in terms of computational complexity.

Constraints Allow to Speed up Computation Process. As already introduced, CARE is capable of including metamodel-specific OCL constraints as well as user constraints, e.g., mapping instructions. Since these constraints represent hard facts, i. e., they narrow the solution space, they are an essential source for efficiently generating valid solutions. Consequently, with many constraints to be exploited in the computation process, CARE exhibits a considerably increased runtime performance. Nevertheless, in certain cases the presence of OCL constraints may lead to an empty solution, if no repair actions for OCL violations are present. Those repair actions have to be defined manually in the current prototype. An investigation of how to automatically derive repair actions for OCL constraints is part of future work.

Ranking Depends on Favored Quality Criteria. As mentioned, CARE provides a default configuration for ranking, which focusses on the prevention of information loss as a quality criterion for the resulting models. This may be undesired, e.g., in case that classes in the metamodel have been deleted and, consequently, corresponding instances should be deleted as well. Thereby, the incorporation of semantic knowledge may be beneficial to recognize semantically related concepts and, thus, rank corresponding solutions higher. Consequently, the user may influence the ranking by providing additional knowledge, which in turn favors different quality criteria.

Runtime Complexity & Scalability. As seen by means of the presented example, the solution space exponentially depends on the size of the metamodels and on the number of non-conforming instances. Referring to the formula for complexity $(c+1)^0$ for the example (cf. Sect. 2), one may see that in case of many non-conforming instances counter-measures are indispensable for efficient computation. Therefore, CARE provides several of such measures including the usage of constraints, which reduce the number of possible solutions and, therefore, heavily improve runtime performance. Another way to alleviate this problem is to reclassify by class, instead of per object (e. g., all State objects become reclassified to the same class), effectively reducing the complexity to be linear, only.

Meta Information Provides a Valuable Source for Selection. In cases, where multiple solution models remain for selection, which all exhibit the same costs according to the provided quality criteria, the generated meta-information in terms of provenance information and statistics represents a valuable source for the user to inspect and distinguish the solutions. Consequently, differences may be determined, to ultimately select the best solution.

In summary, CARE allows to re-establish conformance without comprising additional knowledge, but is especially useful for metamodels with comprehensive constraints, to generate the best solutions enriched by provenance information.

6 Related Work

This section discusses related approaches on detecting and repairing ontological conformance violations. Table 3 compares related approaches as well as our own approach CARE, regarding criteria discussed in Section 1 comprising *tool coupling* and *application scope*, *user involvement*, and provision of *multiple solutions*, as well as criteria related to the three phases of the CARE approach, i. e., ① *detection*, ② *repairing* of constraint violations, and ③ *ranking* of multiple solutions. Approaches are classified with respect to *tight coupling*, i. e., deeply woven into an IDE, or *loose coupling* to a specific modeling environment. *Application scope* indicates the specific or general applicability of the approach, by specifying the kinds of artifacts between which conformance is re-established. *Detection* of constraint violations considers both, the *specification* and the *examination* of constraints to expose their violations. Regarding *repairing* of constraint violations, we investigated generation of *solutions*, checking for *contradictory repairing*, and *user involvement* by examining *user effort* for repairing, e. g., specification of rules, and whether or not interactive *user guidance* is required for finding a valid solution. Approaches generating multiple solutions are examined for automatic *ranking.* Finally, we investigated if meta information by means of *provenance* of the generated solutions is provided.

The examined approaches have been selected due to their capability for fixing inconsistencies in models or re-establishing conformance between models and their metamodels. They are classified according to *tool coupling* into either tight [8, 10, 13, 18, 21] or loosely [1, 3, 15, 23, 27, 29] coupled approaches, whereas the latter are more closely related to the CARE approach, since CARE is not coupled to or woven into a specific tool. The *application scope* ranges from detecting or repairing of constraint violations in UML models [1, 8, 21, 23, 27], over viewpoint synchronization [10], and re-establishing conformance in graph based models [15], to fixing of inconsistencies of models using the OCL related language BeanBag [29], and coupled evolution of metamodels and models [13]. CARE proposes a more generic approach for re-establishing conformance of models to a given Ecore-based metamodel. *Specification* and *examination* for *detecting* of constraint violations is performed by logic programming [1, 3, 10, 21, 23, 27] using different execution engines, graph transformation rules [18], Java [13], or using the language BeanBag [29]. Like most approaches, CARE also bases on

[6] http://www.dlvsystem.com/

App-roach	Tool Coup-ling	Application Scope	① Detection		② Repairing				③ Ranking & Provenance	
			Specification	Examination	Solutions	Non-Contra-dictory	User Involvement			
							Low User Effort	Independence of User Guidance	Ranking	Prove-nance
[1]	Loose	UML MM -> UML model	Prolog Rules	✓ Logic pre-dicates (Praxis)	(Partial) repair plans	✗	✗ Specify cause detection rules	✓ Not required	Not required	✗
[3]	Loose	UML MM -> UML model	Prolog Rules	✓ Logic pre-dicates (Prolog)	✗	n.a.	n.a.	n.a.	n.a.	✗
[8]	Tight	UML model A -> UML model B	Abstract Rule Language	✓ (Model profiler)	Single solution	✓	✓	✗ Pick resolution action	Not required	✗
[10]	Tight	Viewpoint A -> Viewpoint B	ASP	✓ Logic pre-dicates (Smodels)	Multiple solutions	✓	✓	✗ Refine sol-utions iteratively	✗	✗
[13]	Tight	MM -> M co-evolution	Java	✗ (Edapt)	Single solution	✗	✗ Specify custom migration rules	✗ Define coupled evolution steps	Not required	✗
[15]	Loose	Graph based models	Triple Rules (TGG)	✓ Consistency rules	Multiple repair actions	✗	✓ Select repair action	✓ Not required	✗	✗
[18]	Tight	UML MM -> UML model	Graph trans-formation rules	✓ (AGG)	Single solution	✓	-	✗ Pick resolution action iteratively	Not required	✗
[21]	Tight	Repair distributed UML documents	First Order Logic	✓ Logic pre-dicates (xlinkit)	Single repair action	✓	✗ Create repair action	✗ Pick repair actions	Not required	✗
[23]	Loose	UML class diagram A -> UML class diagram B	PDDL	✓ Logic pre-dicates (PDDL)	Only plans	✓	-	✓ Not required	Not required	✗
[27]	Loose	UML model A -> UML model B	Description Logic	✓ Logic pre-dicates (Loom)	Single solution	✓	✗ Specify models in DL	✓ Not required	Not required	✗
[29]	Loose	Model inconsistency fixing	BeanBag program	✓ (BeanBag)	Single solution	✗	✗ Define beanbag program	✓ Not required	Not required	✗
CARE	Loose	Re-establishing M -> MM conformance	ASP	✓ Logic pre-dicates (DLV Solver)	Multiple solutions	✓	✓ Select solution	✓ Not required	✓	✓

Table 3: Comparison of Approaches for Re-Establishing Conformance

logic programming (in terms of ASP), but is unique in using the DLV solver, which allows the usage of weak constraints for ranking, by downgrading non-optimal solutions. Most approaches support *repairing* or re-establishing conformance, either by generating a single solution [8, 13, 18, 21, 27, 29], multiple repair actions [15] or repair solutions [10], or by creating (partial) repair plans [1, 23]. Like only one other approach [10], CARE is able to generate multiple solutions in parallel, from which the user may select the most appropriate one, but is unique with respect to ranking them. Checking for *non-contradictory* repairing is essential, to either restrict the solutions or generate valid repair plans, and is performed by various approaches [8,10,18,21,23,27], as well as by CARE, which computes valid solutions, only. Furthermore, we examined approaches with respect to the required extent of *user involvement*. Several approaches require manual *effort* to enable detection of non-conformance in models [1,21,29], while others rely on interactive *user guidance* for finding a valid solution [8, 10, 13, 18, 21]. Unlike those approaches, CARE does not rely on user interaction, as the provided generic conformance constraints as well as metamodel constraints allow to generate multiple valid solutions, but optionally user knowledge can be incorporated to further improve the generation and ranking of results. In contrast to the examined approaches, which support the generation of multiple, but *unranked* solutions [10, 15], CARE provides built-in heuristics which allow for configuration and, thus, an appropriate *ranking* of the generated solutions. CARE is unique with respect to providing *provenance information* of the generated solution.

In summary, when surveying related work with respect to CARE, one may see that CARE is unique with respect to the combination of (i) loose coupling to a specific tool or modeling environment, (ii) low user effort in the repairing process, and (iii) offering multiple valid and ranked solutions that are further enhanced by provenance information, which is then reported to the user for inspecting the differences between generated solutions.

7 Conclusion & Future Work

In this paper, we have shown, how a violated *ontological conformance* relationship between a model and its metamodel may be re-established based on *conformance constraints*, facilitated by logic programming. In particular, the presented approach supports the *detection of conformance violations* and provides *repair actions* to re-establish this conformance relationship. In contrast to other approaches, not only a single solution, but a *set of ranked* solutions is generated based on several quality criteria exploiting structural and semantic knowledge that is enriched with provenance information and statistics to facilitate inspection by the user.

The current realization shows great potential, but there are several lines of future work. As already mentioned, the transformation to and from ASP is foreseen to be fully automated in the future. In this context, especially the generation of repair actions for OCL violations is of great interest, including the potential for reusing the already implemented repair actions.

Ultimately, given the appropriate transformations, the implementation may be adapted to other domains and technical spaces besides Ecore, e. g., database schemas with according instances. Alternatively, CARE may also be applied for the evolution of interdependent semantic web ontologies with accompanying OWL constraints. Thereby, multi-domain ontologies such as DBpedia may be employed as source for semantic knowledge with the `dlvhex` [9] solver.

References

[1] M. Almeida da Silva, A. Mougenot, X. Blanc, and R. Bendraou. Towards Automated Incon-

sistency Handling in Design Models. In *Proc. of the 22nd Int. Conf. on Advanced Information Systems Engineering*. Springer, 2010.

[2] S. Berger, G. Grossmann, M. Stumptner, and M. Schrefl. Metamodel-Based Information Integration at Industrial Scale. In *Proc. of the 13th Int. Conf. on Model Driven Engineering Languages and Systems*, 2010.

[3] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting Model Inconsistency through Operation-Based Model Construction. In *Proc. of the 30th Int. Conf. on Software Engineering*. ACM, 2008.

[4] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: A Bidirectional and Change Propagating Transformation Language. In *Proc. of the 3rd Int. Conf. on Software Language Engineering*. Springer, 2011.

[5] A. Dahanayake and B. Thalheim. Co-evolution of (Information) System Models. In I. Bider, T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, and R. Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 50 of *Lecture Notes in Business Information Processing*. Springer, 2010.

[6] M. G. de Carvalho, A. H. Laender, M. A. Gonalves, and A. S. da Silva. An Evolutionary Approach to Complex Schema Matching. *Information Systems*, 38(3), 2013.

[7] D. Di Ruscio, L. Iovino, and A. Pierantonio. What is Needed for Managing Co-Evolution in MDE? In *Proc. of the 2nd Int. Workshop on Model Comparison in Practice*. ACM, 2011.

[8] A. Egyed, E. Letier, and A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *Proc. of the 2008 23rd IEEE/ACM Int. Conf. on Automated Software Engineering*. IEEE, 2008.

[9] T. Eiter, G. Ianni, and T. Krennwallner. Answer Set Programming: A Primer. *Reasoning Web. Semantic Technologies for Information Systems*, 2009.

[10] R. Eramo, A. Pierantonio, J. Romero, and A. Vallecillo. Change Management in Multi-Viewpoint System Using ASP. In *Proc. of the Workshop on ODP for Enterprise Computing, WODPEC, 2008*. IEEE, 2008.

[11] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[12] M. Hartung, J. Terwilliger, and E. Rahm. Recent Advances in Schema and Ontology Evolution. In *Schema Matching and Mapping*. Springer, 2011.

[13] M. Herrmannsdoerfer. COPE – A Workbench for the Coupled Evolution of Metamodels and Models. In *Proc. of the 3rd Int. Conf. on Software Language Engineering*. Springer, 2011.

[14] M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In *Proc. of the 3rd Int. Conf. on Software Language Engineering*. Springer, 2011.

[15] A.-T. Körtgen. New Strategies to Resolve Inconsistencies between Models of Decoupled Tools. In *Proc. of 3rd Workshop on Living with Inconsistencies in Software Development*, 2010.

[16] T. Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4), Dec 2006.

[17] V. Lifschitz. What is Answer Set Programming? In *Proc. of the AAAI Conf. on Artificial Intelligence*, 2008.

[18] T. Mens, R. Straeten, and M. D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *Proc. of the 9th Int. Conf. on Model Driven Engineering Languages and Systems*. Springer, 2006.

[19] M. Milanović, D. Gašević, A. Giurca, G. Wagner, and V. Devedžić. On Interchanging Between OWL/SWRL and UML/OCL. In *Proceedings of 6th Workshop on OCL for (Meta-) Models in Multiple Application Domains*, 2006.

[20] L. Moreau et al. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, 2011.

[21] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *Proc. of 25th Int. Conf. on Software Engineering*. IEEE, 2003.

[22] Object Management Group. Meta Object Facility (MOF) 2 Core Specification. www.omg.org/spec/MOF/2.4.1/PDF, 2011.

[23] J. P. Puissant, T. Mens, and R. Van Der Straeten. Resolving Model Inconsistencies with Automated Planning. In *Proc. of the 3rd Workshop on Living with Inconsistencies in Software Development*, 2010.

[24] A. Queralt and E. Teniente. Reasoning on UML Class Diagrams with OCL Constraints. In *Proc. of the 25th Int. Conf. on Conceptual Modeling*. Springer, 2006.

[25] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10, Dec 2001.

[26] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[27] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using Description Logic to Maintain Consistency between UML Models. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*. Springer, 2003.

[28] M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, and G. Kappel. On using Inplace Transformations for Model Co-evolution. In *Proc. of the 2nd Int. Workshop on Model Transformation with ATL at TOOLS*, 2010.

[29] Y. Xiong et al. Supporting Automatic Model Inconsistency Fixing. In *Proc. of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of Software Engineering*. ACM, 2009.