Sound Merging and Differencing for Class Diagrams

Uli Fahrenberg, Mathieu Acher, Axel Legay, Andrzej Wasowski

Sound Merging and Differencing for Class Diagrams

Uli Fahrenberg[1], Mathieu Acher[1,2], Axel Legay[1], and Andrzej Wąsowski[3*]

[1] IRISA / Inria Rennes, France
[2] University of Rennes 1, France
[3] IT University of Copenhagen, Denmark

Abstract. Class diagrams are among the most popular modeling languages in industrial use. In a model-driven development process, class diagrams evolve, so it is important to be able to assess differences between revisions, as well as to propagate differences using suitable merge operations. Existing differencing and merging methods are mainly syntactic, concentrating on edit operations applied to model elements, or they are based on sampling: enumerating some examples of instances which characterize the difference between two diagrams. This paper presents the first known (to the best of our knowledge) automatic model merging and differencing operators supported by a formal semantic theory guaranteeing that they are semantically sound. All instances of the merge of a model and its difference with another model are automatically instances of the second model. The differences we synthesize are represented using class diagram notation (not edits, or instances), which allows creation of a simple yet flexible algebra for diffing and merging. It also allows presenting changes comprehensively, in a notation already known to users.

1 Introduction

Model management is an essential activity in a model-driven development process. Numerous tools exist to visualize, validate, transform, refactor, compute differences, or merge models and structured data. The basic management operations can be combined to realize complex maintenance and design tasks. In this paper, we consider merging and differencing of models—two crucial management operations—for class diagrams, the most popular modeling language used in the industry [12]. Class diagrams are used to create domain models, structural system models, and lower level design models. Class diagrams also serve as meta-models, or abstract syntax types, in implementation of domain specific languages (DSLs).

Merging. When a single model is not sufficient to capture all the aspects of a problem or a system, engineers have to merge several models to produce a single integrated model [11, 13, 19, 20, 23, 25]. Model merging also arises when factoring out commonalities of different variant models, e.g., following a product line approach (e.g., see [1, 21, 22]).

[*] Supported by The Danish Council for Independent Research under *Sapere Aude* project VARIETE

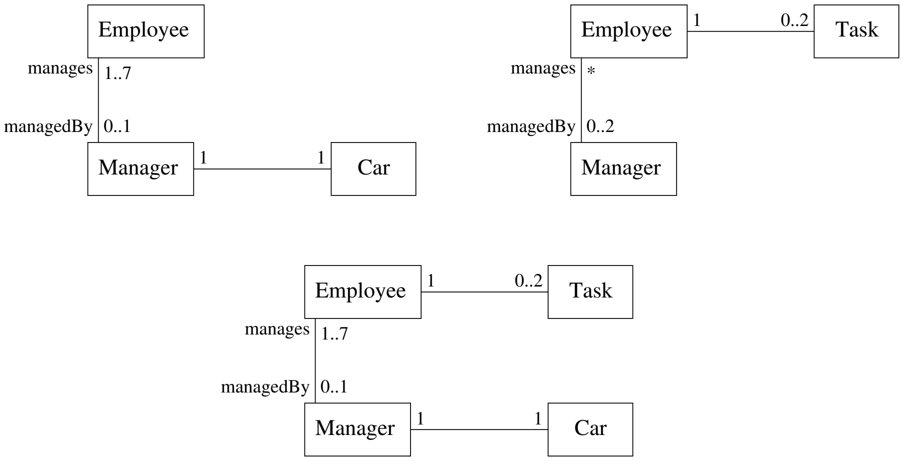


Fig. 1. Two simple class diagrams (above) and their merge (below). Note that the object models of the merge are precisely the ones which model both individual aspects: here, the merge is both sound and complete.

Differencing. Models naturally evolve through editing operations (e.g., adding an inheritance relationship in a class diagram) for refactoring or extending a system. Computing differences of two revised models has applications in comprehension of an evolution, identification of instances not supported by the two models [6,14], etc. Merging and differencing operations can be combined when engineers have to propagate differences from one version of the model to another, as in the case of revision control systems.

Key requirements. We highlight two requirements that arise when merging or differencing models. In many engineering scenarios, the result of a merge or difference is subject to *analysis*: automated tools can ensure that certain instances are still present in the composed diagram; engineers (humans) can visualize the result and therefore understand the evolution of the system. Another important requirement is that the result of a merge (resp. difference) *advances the design of a system*, so it produces a new model that can later be used in construction. Again, both automated tools and manual activities can operate over this model. For instance, code can be generated from a class diagram; engineers can relate the class diagram to other modeling artefacts or specify a transformation.

The two requirements impact the design of merging and differencing operations. First, it is beneficial that a merge (resp. difference) of two class diagrams should be a class diagram—in line with the vision exposed in [10] that a difference between models should be a model. This allows engineers to visualize the difference and to manipulate it using the usual tools for working with models. Second, merge (resp. difference) operations should be ideally both sound and complete, to enable further precise analysis by automated tools. In particular, an *unsound* merge operator can authorize objects (instances) actually not conformant to the merged diagrams. Figure 1 introduces a simple example of a merge between two class diagrams. The result (below) *is* a class diagram, and the merge is both sound and complete. (Note that we assume an open-world semantics, which is consistent with the view semantics used by most modeling tools, i.e. it attempts to compute a minimal model such that the two original models can be derived as its views.)

As further discussed in Sect. 2, existing merging or differencing methods, though extremely useful in practice, do not meet the two requirements. First, syntactic methods suffer from possible lack of soundness, since they do not take the semantics into account. Second, enumerative semantic methods (e.g., [15–17]) only synthesize a finite set of instance and are thus not complete. In both methods, the result is no longer a class diagram, precluding an algebraic combination of operations or a direct manipulation by tools and engineers.

We follow the methodology presented by us in an earlier work [10]—we develop the difference operator systematically, via a number of well-defined formal definition steps. We believe that these steps yield a substantial understanding of what the differences between diagrams can be (unlike methods that simply encode a lot of knowledge in a large constraint system solved by an external solver). This knowledge can later be used to advance language design for structural modeling.

The contributions of the paper include:

- A *concise formal semantics* for the core language of class diagrams (concise and mathematically mature definitions of such are surprising hard to find). The semantics relies purely on set theory, which is crucial for avoiding dependencies on description logics, reasoners and proof systems.
- A formalisation of the standard (conjunctive) *merge* operator used to compute the conjunction of class diagrams (by "putting them on the same page"). To the best of our knowledge, no previous such formalisations exist; using our formalization, we can show that merge is always, automatically, *sound*.
- An intuitive notion of *subtyping* which is closely related to the merge operator (which is the greatest lower bound for subtyping), and a corresponding notion of a *disjunctive merge*, which computes disjunctions of class diagrams (and is the lowest upper bound for subtyping). Using our formalization, we show that disjunctive merge is semantically *complete*.
- A *compositional algebra* for class diagrams: We can show that using only (conjunctive) merge, all class diagrams can be constructed from a few *elementary* class diagrams. Together with subtyping, this gives an algebra which allows for high-level computations with class diagrams.
- A *difference* operator which computes, in a precise sense, the best possible approximation to an inverse to conjunctive merge. Our difference operator is semantically sound, computable entirely automatically, and produces finite syntactic descriptions of infinite (semantic) differences.

The intended audience are researchers interested in semantics of structural modeling, evolution of meta-models and builders of differencing or merging tools.

We proceed by discussing related work in Sect. 2. Section 3 introduces a simple abstract syntax for class diagrams, along with subtyping and compositions. Sect. 4 defines differencing. Both sections work at syntactic level. In Sect. 5, we justify these constructions by showing that they are semantically sound. We introduce a concise, set-based semantics for class diagrams and show that our operators respect it. We conclude in Sect. 6, indicating possible variants and extensions.

2 Related Work

Our objective is to represent merges and differences of two class diagrams *as a class diagram*, overcoming limitations of enumerative and syntactical methods in terms of soundness, completeness, and further exploitation by engineers. We achieve this by developing syntactic methods (with sound theoretical basis), which guarantee representation of difference in one description as a diagram.

Limitations of existing syntactic approaches. A syntactic approach to differencing operates purely by manipulating syntactic elements of diagrams, mostly without paying attention whether these manipulations are sound with respect to semantics of the language (for example that they do not ignore some instances).

Syntactic differencing methods are extremely useful in practice, for example, users can visualize a set of syntactic edit operations (e.g., create, modify, delete [3]) and easily understand the evolution of two models. Most of the studies in the field of model differencing (see, e.g., [6]) present syntactic differencing at either the concrete or the abstract syntax level. However, syntactic methods are inherently incomplete and unsound. They may not be able to expose and represent the semantic (meaningful) differences between two versions of a class diagram. They can also report false positives by operating at the syntactical level. The incompleteness and unsoundness can both disturb automated analysis and a further exploitation by an engineer.

Limitations of enumerative approaches. As argued in some papers and illustrated for some formalisms [2,7,10,15–17], models that are syntactically very similar may induce very different semantics, and a list of differences should be best addressed *semantically.* Recently, semantic approaches have been proposed which enumerate some examples of *instances* of one model that are not instances of the other [17]. For instance, Maoz *et al.* tackle the problem of semantic model differencing, specifically for class and activity diagrams [15, 16]. The *cddiff* operator for class diagrams [16] computes diff witnesses using the Alloy Analyzer, a solver for relational logic with transitive closure. For the *addiff* operator for activity diagrams, they present algorithms that take as input activity diagrams [15]. The advantage of enumerative methods is that they operate at the semantic level and are sound by construction: no false positives can be reported. However they are clearly incomplete since only a small, finite number of examples can be synthesized. In fact, the set of instances in the difference might well be infinite. A related problem is that engineers cannot visualize and manipulate the (infinite) set of witnesses—a concrete and compact representation is missing. It also precludes an algebraic combination of merging and differencing operations, as in the case of versioning control systems (see Fig. 3, page 10 for more details).

Other approaches and existing tools. Model *matching* is another important related problem [6, 7, 9]. Numerous algorithms and techniques are already integrated in model-based tools (e.g., see [14, 19, 23]). For simplicity the theoretical framework we develop assume a basic matching strategy (based on names, see next section). Numerous "diff" or merging tools (e.g., EMF Diff[4], Kompose [11, 20], Epsilon [13], UMLDiff [25], TReMer+ [19, 23], etc.) offer the means to specify user directives (using a specific language or an API). Users can override (customize) the default behaviour of the merging or differencing algorithm and thus handle the semantics of the models. However, the manual specification, if not properly defined, does not guarantee semantic properties and can lead to unsound or incomplete merging (resp. differencing). The objective of the paper is precisely to study the soundness and completeness of operations that can be incorporated into modeling tools. Our approach is both algebraically and semantically justified: to the best of our knowledge, no previous effort guarantees such properties of merging and differencing of class diagrams. Any matching strategy that defines an equivalence on the space of class names could be naturally incorporated in our framework.

3 Compositional Algebra of Class Diagrams

We start by introducing an abstract syntax for class diagrams. We remark that the operators defined in this section are entirely syntactical; no reference to the semantics of class diagrams is made. The same holds for the properties we expose: they are proven at the syntactic level. In Sect. 5 we will introduce a semantics for class diagrams and show that our operators are semantically sound, but their properties as shown in the present section are completely independent of any semantics one wishes to give to class diagrams.

Let $\mathcal{N}$ be the set of all finite unions of finite or unbounded intervals of natural numbers (including the empty set of intervals). $\mathcal{N}$ is closed under union, intersection and complementation; let $\neg A = \mathbb{N} \setminus A$ be the complement of $A \in \mathcal{N}$.

3.1 Abstract syntax

Let Σ_c, Σ_a and Σ_e be disjoint infinite sets of names, for classes, associations and association ends, respectively.

Definition 1. *A* class diagram *is a tuple* $\mathcal{C} = (\mathsf{cla}, \mathsf{asc}, \mathsf{gen}, \mathsf{disj}, \mathsf{ccard}, \mathsf{aends}, \mathsf{acards})$ *consisting of*

(i) *A finite set* $\mathsf{cla} \subseteq \Sigma_c$ *of* classes*,*
(ii) *A finite set* $\mathsf{asc} \subseteq \Sigma_a$ *of* associations*,*
(iii) *A reflexive transitive relation* $\mathsf{gen} \subseteq \mathsf{cla} \times \mathsf{cla} \cup \mathsf{asc} \times \mathsf{asc}$ *capturing* generalizations *between classes and between associations,*
(iv) *An irreflexive symmetric relation* $\mathsf{disj} \subseteq \mathsf{cla} \times \mathsf{cla}$ *representing class* disjointness *constraints,*

[4] http://eclipse.org/diffmerge/

(v) *A mapping* $\mathsf{ccard} : \mathsf{cla} \to \mathcal{N}$ *encoding class* cardinality *constraints,*

(vi) *A partial function* $\mathsf{aends} : \mathsf{asc} \to \Sigma_e \rightharpoonup \mathsf{cla}$ *mapping each association to its endpoints, and*

(vii) *A partial function* $\mathsf{acards} : \mathsf{asc} \to \Sigma_e \rightharpoonup \mathcal{N}$ *mapping each association to its endpoint cardinality constraints.*

Also $|\mathsf{dom}(\mathsf{aends}(a))| = 2$ *and* $\mathsf{dom}(\mathsf{acards}(a)) = \mathsf{dom}(\mathsf{aends}(a))$ *for all* $a \in \mathsf{asc}$.

Note that we handle generalizations of both classes and associations; this is common in modern modeling approaches. In the usual concrete syntax of class diagrams, the generalization relation is essentially transitively reduced, but in (iii) we require it to be transitively closed, in order to simplify presentation. Given that the sets of classes and associations are finite, switching between both viewpoints is just a technicality. Another common assumption in class modeling is that two classes which do not share a common subclass are disjoint. This default assumption does not work well with the open-world semantics that we will build up in this paper. It would make the semantics non-monotonic—adding a shared subclass would relax constraints on instances. To prevent this, we prefer to add, in (iv), an explicit representation of binary disjointness constraints in the abstract syntax. Again, switching between explicit and implicit disjointness constraints is just a technicality, but it will allow us to simplify presentation later on. The last condition means that we only consider binary associations. With this in mind, the function aends (vi) maps the two association ends (or, more precisely, their names) of each association to their classes. Similarly, the function acards (vii) associates cardinality constraints to association ends.

As mentioned we will use an open-world semantics, which can be summarized by the slogan that *anything which is not forbidden is allowed.* So, intuitively, there will be no semantic difference between a class c that does not appear in a given class diagram, and one which does, but has unrestricted cardinality. Formally, we call a class $c \in \mathsf{cla}$ in some class diagram $\mathcal{C}$ *restricted* if $\mathsf{ccard}(c) \neq \mathbb{N}$, and let $\mathsf{rcla} \subseteq \mathsf{cla}$ denote the subset of restricted classes.

3.2 Merge

Ultimately, we want to create class diagrams by composing smaller chunks of them. This reflects the practice of modeling with views, as supported by many modeling tools (for example Eclipse Modeling Framework, Papyrus, or IBM Rational Modeler). Users of such tools work with projections of one large single model represented implicitly in a unified syntax tree. From the users' perspective it may often appear that they work with a number of diagrams that are unified (composed) as if they were *put together on the same page*, merging entities (e.g., classes) that have the same name. In the following, we propose a conjunctive *merge* operator, written $\owedge$, that composes two diagrams, as if they were put on the same page, interpreted as views of the same underlying model.

Due to syntactic restrictions, it is not possible for us to merge diagrams that have the same association with different association ends. Since we only allow one name per endpoint in abstract syntax, there is no way to merge different names. This is consistent with the behavior of common modeling tools, where different views of the same association always maintain the same end points.

Definition 2. *Two class diagrams* $\mathcal{C}_1$*,* $\mathcal{C}_2$ *are said to be* composable *if it holds for all* $a \in \mathsf{asc}_1 \cap \mathsf{asc}_2$ *that* $\mathsf{aends}_1(a) = \mathsf{aends}_2(a)$*.*

Definition 3. *The* (conjunctive) merge *of two composable class diagrams* $\mathcal{C}_1$*,* $\mathcal{C}_2$ *is* $\mathcal{C}_{\owedge} = \mathcal{C}_1 \owedge \mathcal{C}_2$ *defined as follows:*

- $\mathsf{cla}_{\owedge} = \mathsf{cla}_1 \cup \mathsf{cla}_2$, $\mathsf{asc}_{\owedge} = \mathsf{asc}_1 \cup \mathsf{asc}_2$, $\mathsf{gen}_{\owedge} = (\mathsf{gen}_1 \cup \mathsf{gen}_2)^*$, $\mathsf{disj}_{\owedge} = \mathsf{disj}_1 \cup \mathsf{disj}_2$,
- $\mathsf{aends}_{\owedge} = \mathsf{aends}_1 \cup \mathsf{aends}_2$*, and*

$$\mathsf{ccard}_{\owedge}(c) = \begin{cases} \mathsf{ccard}_1(c) & \text{if } c \in \mathsf{cla}_1 \setminus \mathsf{cla}_2, \\ \mathsf{ccard}_2(c) & \text{if } c \in \mathsf{cla}_2 \setminus \mathsf{cla}_1, \\ \mathsf{ccard}_1(c) \cap \mathsf{ccard}_2(c) & \text{if } c \in \mathsf{cla}_1 \cap \mathsf{cla}_2, \end{cases}$$

$$\mathsf{acards}_{\owedge}(a)(e) = \begin{cases} \mathsf{acards}_1(a)(e) & \text{if } a \in \mathsf{asc}_1 \setminus \mathsf{asc}_2, \\ \mathsf{acards}_2(a)(e) & \text{if } a \in \mathsf{asc}_2 \setminus \mathsf{asc}_1, \\ \mathsf{acards}_1(a)(e) \cap \mathsf{acards}_2(a)(e) & \text{if } a \in \mathsf{asc}_1 \cap \mathsf{asc}_2 \end{cases}$$

for all $c \in \mathsf{cla}_{\owedge}$*,* $a \in \mathsf{asc}_{\owedge}$*, and* $e \in \mathsf{dom}(\mathsf{aends}(a))$*.*

Intuitively, the above merging attempts to approximate *conjunction* of class diagrams. Due to our open-world semantics ("anything which is not mentioned is unrestricted"), we have to apply a *disjunction* to the syntactic elements (classes, associations, etc.) to get a conjunctive merge. The merge in Fig. 1 in the introduction gives a simple example of the operation.

3.3 Subtyping

With the above definitions we have gathered enough structure to propose a definition of *subtyping* between two class diagrams:

Definition 4. *For class diagrams* $\mathcal{C}_1$*,* $\mathcal{C}_2$*, we say that* $\mathcal{C}_1$ *(syntactically)* refines $\mathcal{C}_2$*, denoted* $\mathcal{C}_1 \leq \mathcal{C}_2$*, if all the following conditions hold:*

(i) $\mathsf{cla}_1 \supseteq \mathsf{rcla}_2$, $\mathsf{asc}_1 \supseteq \mathsf{asc}_2$
($\mathcal{C}_1$ is an extension of $\mathcal{C}_2$)

(ii) $\mathsf{gen}_1 \supseteq \mathsf{gen}_2$, $\mathsf{disj}_1 \supseteq \mathsf{disj}_2$
(generalization and disjointness constraints are inherited)

(iii) $\mathsf{ccard}_1(c) \subseteq \mathsf{ccard}_2(c)$ *for all* $c \in \mathsf{rcla}_2$
(class cardinalities are restricted)

(iv) $\mathsf{aends}_1(a) = \mathsf{aends}_2(a)$ *for all* $a \in \mathsf{asc}_2$
(association ends are preserved)

(v) $\mathsf{acards}_1(a)(e) \subseteq \mathsf{acards}_2(a)(e)$ *for all* $a \in \mathsf{asc}_2$ *and all* $e \in \mathsf{dom}(\mathsf{acards}_2(a))$
(association cardinalities are restricted)

Due to the open-world semantics, subtyping on syntactic elements becomes reversed subset inclusion. Also, we have to use *restricted* classes in the inclusion $\mathsf{cla}_1 \supseteq \mathsf{rcla}_2$ above; otherwise one could easily find subtypings which were *not sound*, i.e., where the subtype admits models which the supertype does not.

Our selections of subtyping and merging are strongly related, in the sense that $\owedge$ is the *greatest lower bound* for $\leq$. In practice, this means that the merge does indeed "behave like a conjunction".

Theorem 1. *For all class diagrams $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{D}$ with $\mathcal{C}_1$ and $\mathcal{C}_2$ composable, $\mathcal{D} \leq \mathcal{C}_1 \owedge \mathcal{C}_2$ iff $\mathcal{D} \leq \mathcal{C}_1$ and $\mathcal{D} \leq \mathcal{C}_2$.*

We believe that this compatibility of subtyping and merging is of fundamental importance. Subtyping captures the intuitive notion of constraining the set of instances during modeling. A merge operator considered without subtyping can create merges that admit far fewer, or many more, instances than the user would expect. Introducing the above subtyping preorder on diagrams will later allow us to follow the method presented in [10] for defining differences. Intuitively, we need an order because notions of difference, distance and order are intimately related. It is surprising that so few works on merging and differencing structural models consider subtypes or other orderings (unlike for behavioral models, e.g. [8]).

Theorem 1 immediately entails that the merge operator has the core properties expected of composition, viz. commutativity and associativity. Also, merge is monotonic with respect to the subtyping order, or, in other words, *subtyping is compositional.* This is a highly desirable property that introduces some regularity in the framework.

Theorem 2. *The $\owedge$ operator is commutative and associative, and for all class diagrams $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{D}_2$, $\mathcal{D}_2$ with $\mathcal{C}_1$ and $\mathcal{D}_1$ composable and $\mathcal{C}_2$ and $\mathcal{D}_2$ composable, $\mathcal{C}_1 \leq \mathcal{C}_2$ and $\mathcal{D}_1 \leq \mathcal{D}_2$ imply $\mathcal{C}_1 \owedge \mathcal{D}_1 \leq \mathcal{C}_2 \owedge \mathcal{D}_2$.*

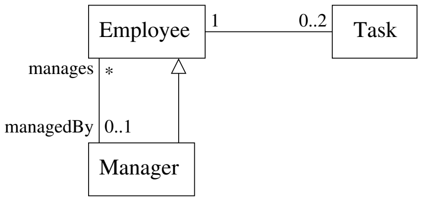
3.4 Class Diagram Algebra

Compositionality allows us to develop an algebra for class diagrams, using a few elementary diagrams and the composition operator $\owedge$. What we obtain is a small structural modeling calculus that is of interest by itself — class diagrams can be written concisely, in a manner that is friendly to a linear mathematically oriented text, unlike the concrete syntax representation of class diagrams, and unlike the rather unwieldy abstract syntax presented in Def. 1.

The calculus is built around a set of elementary class diagrams which are merged to obtain bigger structures. The elementary diagrams are as follows:

- $\top = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$: the empty class diagram that admits all object models;
- $\langle c^n \rangle$, for $c \in \Sigma_c$ and $n \in \mathcal{N}$: the class diagram with $\mathsf{cla} = \{c\}$, $\mathsf{asc} = \emptyset$, $\mathsf{gen} = \{(c, c)\}$, $\mathsf{disj} = \emptyset$, $\mathsf{ccard}(c) = n$, and $\mathsf{aends} = \mathsf{acards} = \emptyset$;
- $\langle c_1 \overset{e_1}{\underset{n_1}{\text{———}}} a \overset{e_2}{\underset{n_2}{\text{———}}} c_2 \rangle$, for $c_1, c_2 \in \Sigma_c$, $e_1, e_2 \in \Sigma_e$ and $n_1, n_2 \in \mathcal{N}$: the class diagram with $\mathsf{cla} = \{c_1, c_2\}$, $\mathsf{asc} = \{a\}$, $\mathsf{gen} = \{(c_1, c_1), (c_2, c_2)\}$, $\mathsf{disj} = \emptyset$, $\mathsf{ccard}(c_1) = \mathsf{ccard}(c_2) = \mathbb{N}$, $\mathsf{dom}(\mathsf{aends}(a)) = \{e_1, e_2\}$, $\mathsf{aends}(a)(e_1) = c_1$, $\mathsf{aends}(a)(e_2) = c_2$, $\mathsf{acards}(a)(e_1) = n_1$, and $\mathsf{acards}(a)(e_2) = n_2$;

$$\top \qquad \langle c^n\rangle \qquad \langle c_1 \overset{e_1}{\underset{n_1}{\text{——}}} a \overset{e_2}{\underset{n_2}{\text{——}}} c_2\rangle \qquad \langle c_1 \mathrel{-\!\!\!-\!\!\rhd} c_2\rangle \qquad \langle c_1 \mathrel{-\!/\!/\!-} c_2\rangle$$



$$\begin{gathered}\langle \mathrm{E}^{\mathbb{N}}\rangle \owedge \langle \mathrm{T}^{\mathbb{N}}\rangle \owedge \langle \mathrm{M}^{\mathbb{N}}\rangle \owedge \\ \langle \mathrm{E} \overset{}{\underset{\{1\}}{\text{——}}}\overset{}{\underset{\{0,1,2\}}{\text{——}}} \mathrm{T}\rangle \owedge \langle \mathrm{E} \overset{\mathrm{ms}}{\underset{\mathbb{N}}{\text{——}}}\overset{\mathrm{mB}}{\underset{\{0,1\}}{\text{——}}} \mathrm{M}\rangle \\ \owedge\langle \mathrm{M} \mathrel{-\!\!\!-\!\!\rhd} \mathrm{E}\rangle \\ \owedge\langle \mathrm{E} \mathrel{-\!/\!/\!-} \mathrm{T}\rangle \owedge \langle \mathrm{M} \mathrel{-\!/\!/\!-} \mathrm{T}\rangle\end{gathered}$$

Fig. 2. The five types of elementary class diagrams (above), a simple class diagram in concrete syntax (below left), and its decomposition into elementary class diagrams (below right, where we have abbreviated names).

- $\langle c_1 \mathrel{-\!\!\!-\!\!\rhd} c_2\rangle$, for $c_1, c_2 \in \Sigma_c$: the class diagram with $\mathsf{cla} = \{c_1, c_2\}$, $\mathsf{asc} = \emptyset$, $\mathsf{gen} = \{(c_1, c_1), (c_2, c_2), (c_1, c_2)\}$, $\mathsf{disj} = \emptyset$, $\mathsf{ccard}(c_1) = \mathsf{ccard}(c_2) = \mathbb{N}$, and $\mathsf{aends} = \mathsf{acards} = \emptyset$;
- $\langle c_1 \mathrel{-\!/\!/\!-} c_2\rangle$, for $c_1, c_2 \in \Sigma_c$: the class diagram with $\mathsf{cla} = \{c_1, c_2\}$, $\mathsf{asc} = \emptyset$, $\mathsf{gen} = \{(c_1, c_1), (c_2, c_2)\}$, $\mathsf{disj} = \{(c_1, c_2), (c_2, c_1)\}$, $\mathsf{ccard}(c_1) = \mathsf{ccard}(c_2) = \mathbb{N}$, and $\mathsf{aends} = \mathsf{acards} = \emptyset$.

However simple, the above language of class diagrams is quite powerful—it is fully expressive in the sense that it can express every finite diagram as a finite composition; see also Fig. 2 for an example:

Theorem 3. *Every class diagram can be written as a finite conjunctive merge of elementary class diagrams.*

4 Difference and Disjunctive Merge

We enrich our algebra with two more operators, difference and disjunctive merge. The difference operator, which is a formal adjoint to the conjunctive merge, will have the property that merging a class diagram $\mathcal{C}_1$ with a difference $\mathcal{C}_2 \oslash \mathcal{C}_1$ is semantically *sound*, i.e., the composition $\mathcal{C}_1 \owedge (\mathcal{C}_2 \oslash \mathcal{C}_1)$ is a subtyping of $\mathcal{C}_2$. Later this will translate to a perhaps more intuitive semantic property that the difference does not admit too many instances; adding them to instances of diagram $\mathcal{C}_1$ still obeys the constraints of $\mathcal{C}_2$.

Definition 5. *The* difference *of two composable class diagrams $\mathcal{C}_1$, $\mathcal{C}_2$ is $\mathcal{C}_\oslash = \mathcal{C}_2 \oslash \mathcal{C}_1$ defined as follows:*

- $\mathsf{cla}_\oslash = \mathsf{cla}_2$, $\mathsf{asc}_\oslash = \mathsf{asc}_2 \setminus \mathsf{asc}_1$, $\mathsf{disj}_\oslash = \mathsf{disj}_2 \setminus \mathsf{disj}_1$, *and*

$$\mathsf{gen}_\oslash = \left(\bigcap\{r \subseteq \mathsf{cla}_2 \times \mathsf{cla}_2 \mid (\mathsf{gen}_1 \cup r)^* \supseteq \mathsf{gen}_2\}\right)^*,$$

$$\begin{tikzcd}
& \mathcal{C} \arrow[dd] & \le & \mathcal{D} \arrow[dd] & \\
\mathcal{D}' \oslash \mathcal{D} \arrow[r] & {} & & {} & \mathcal{D}' \oslash \mathcal{D} \arrow[l] \\
& \mathcal{C} \owedge (\mathcal{D}' \oslash \mathcal{D}) & \le & \mathcal{D}' &
\end{tikzcd}$$

Fig. 3. Use of difference for automatic evolving of subtypes: If $\mathcal{C}$ is a subtype of $\mathcal{D}$, then $\mathcal{C}$ merged with the difference $\mathcal{D}' \oslash \mathcal{D}$ is a subtype of $\mathcal{D}'$.

– $\mathsf{aends}_\oslash = \mathsf{aends}_2 \setminus \mathsf{aends}_1$, *and*

$$\mathsf{ccard}_\oslash(c) = \begin{cases} \mathsf{ccard}_2(c) & \textit{if } c \notin \mathsf{cla}_1, \\ \mathsf{ccard}_2(c) \cup \neg \mathsf{ccard}_1(c) & \textit{otherwise}, \end{cases}$$

$$\mathsf{acards}_\oslash(a)(e) = \begin{cases} \mathsf{acards}_2(a)(e) & \textit{if } a \notin \mathsf{asc}_1, \\ \mathsf{acards}_2(a)(e) \cup \neg \mathsf{acards}_1(a)(e) & \textit{otherwise} \end{cases}$$

for all $c \in \mathsf{cla}_\oslash$, $a \in \mathsf{asc}_\oslash$, *and* $e \in \mathsf{dom}(\mathsf{aends}(a))$.

Let us give some intuition about the rather complicated formula for $\mathsf{gen}_\oslash$: Like ordinary set difference, e.g. in $\mathsf{asc}_\oslash = \mathsf{asc}_2 \setminus \mathsf{asc}_1$, is an adjoint to set union, what is on the right-hand side of the $\mathsf{gen}_\oslash$ formula is an adjoint to *transitive union of transitive relations*. That is, $\mathsf{gen}_\oslash$ is the *smallest* transitive relation for which $(\mathsf{gen}_\oslash \cup \mathsf{gen}_1)^* \supseteq \mathsf{gen}_2$. The next theorem states the fundamental property of the difference operator.

Theorem 4. *For all class diagrams* $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$, $\mathcal{C}_1 \owedge \mathcal{C}_2 \le \mathcal{C}_3$ *iff* $\mathcal{C}_1 \le \mathcal{C}_3 \oslash \mathcal{C}_2$.

This means that $\mathcal{C}_3 \oslash \mathcal{C}_2$ is the *most permissive* class diagram for which $(\mathcal{C}_3 \oslash \mathcal{C}_2) \owedge \mathcal{C}_2 \le \mathcal{C}_3$ still holds; in that sense, $\oslash$ is the *natural diff operator* induced by $\owedge$. In this sense, $\oslash$ is the most precise difference operator which can be soundly represented using the class diagram syntax defined in this paper. Since the language elements that we omit do not deal with restricting those that we include, there is no hope that using a richer selection of language elements from the standard set can lead to a better operator. One could, instead, resort to using Object Constraint Language to obtain more precise differencing.

Example 1. Consider a situation as presented in Fig. 3. We have a class diagram $\mathcal{C}$ which is a subtype of another, $\mathcal{D}$. Now the diagram $\mathcal{D}$ is evolved, e.g. by adding more classes or streamlining its associations, into a new class diagram, $\mathcal{D}'$. Our abstract properties of composition and difference now ensure that $\mathcal{C}$ can be evolved to a subtyping of $\mathcal{D}'$, *automatically*, by merging it with $\mathcal{D}' \oslash \mathcal{D}$:

$$\mathcal{C} \owedge (\mathcal{D}' \oslash \mathcal{D}) \le \mathcal{D} \owedge (\mathcal{D}' \oslash \mathcal{D}) \le \mathcal{D}' \qquad \square$$

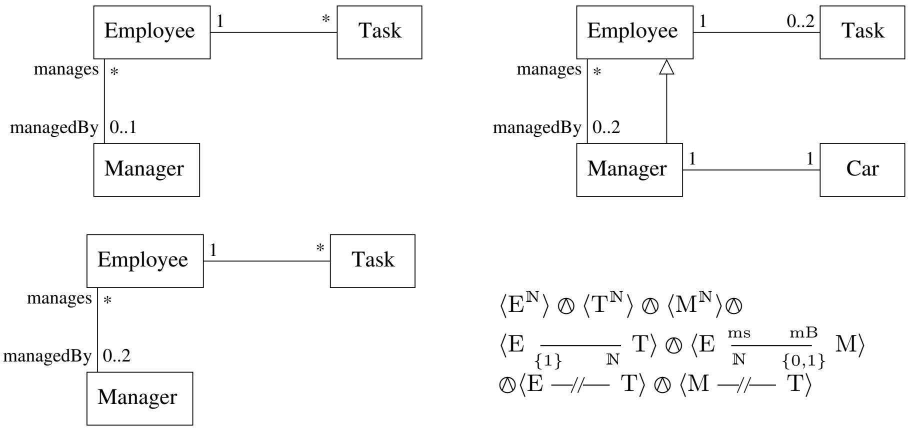

Employee
Task
1
*
manages
*
managedBy
0..1
Manager
Employee
Task
1
0..2
manages
*
managedBy
0..2
Manager
Car
1
1
Employee
Task
1
*
manages
*
managedBy
0..2
Manager
$\langle E^{\mathbb{N}}\rangle \owedge \langle T^{\mathbb{N}}\rangle \owedge \langle M^{\mathbb{N}}\rangle \owedge$
$\langle E \underset{\{1\}\qquad\mathbb{N}}{\overline{\qquad\quad}} T\rangle \owedge \langle E \overset{\mathrm{ms}\qquad\mathrm{mB}}{\underset{\mathbb{N}\qquad\{0,1\}}{\overline{\qquad\quad}}} M\rangle$
$\owedge\langle E \text{—//—} T\rangle \owedge \langle M \text{—//—} T\rangle$

Fig. 4. A disjunctive merge of two diagrams (top) presented in concrete syntax (bottom left) and as a composition of elementary diagrams (bottom right). The overlap between "Manager" and "Employee" seen in the abstract syntax to the right (no disjointness constraint) is not visible in the concrete syntax on the left. The classes can overlap due to the generalization in the rightmost input diagram.

We turn now to another kind of merge operator that can also be induced by subtyping: a "merge in union mode", the *least upper bound* for subtyping:

Definition 6. *The* disjunctive merge *of two composable class diagrams* $\mathcal{C}_1$, $\mathcal{C}_2$ *is* $\mathcal{C}_{\ovee} = \mathcal{C}_1 \ovee \mathcal{C}_2$ *defined as follows:*

$$\mathsf{cla}_{\ovee} = \mathsf{cla}_1 \cap \mathsf{cla}_2, \qquad \mathsf{asc}_{\ovee} = \mathsf{asc}_1 \cap \mathsf{asc}_2, \qquad \mathsf{gen}_{\ovee} = \mathsf{gen}_1 \cap \mathsf{gen}_2$$
$$\mathsf{disj}_{\ovee} = \mathsf{disj}_1 \cap \mathsf{disj}_2, \qquad \mathsf{ccard}_{\ovee}(c) = \mathsf{ccard}_1(c) \cup \mathsf{ccard}_2(c)$$
$$\mathsf{aends}_{\ovee} = \mathsf{aends}_1 \cap \mathsf{aends}_2, \qquad \mathsf{acards}_{\ovee}(a)(e) = \mathsf{acards}_1(a)(e) \cup \mathsf{acards}_2(a)(e)$$

for all $c \in \mathsf{cla}_{\ovee}$*,* $a \in \mathsf{asc}_{\ovee}$*, and* $e \in \mathsf{dom}(\mathsf{aends}(a))$*.*

Note the "contravariance" again: disjunctive merge becomes a conjunction of syntactic elements. As expected, $\ovee$ is least upper bound for $\leq$:

Theorem 5. *For all class diagrams* $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{D}$ *with* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *composable,* $\mathcal{C}_1 \ovee \mathcal{C}_2 \leq \mathcal{D}$ *iff* $\mathcal{C}_1 \leq \mathcal{D}$ *and* $\mathcal{C}_2 \leq \mathcal{D}$*.*

Example 2. Figure 4 shows two variants of the simple class diagram from Fig. 2 together with their disjunctive merge. We see that $\ovee$ extracts precisely the *common features* of the two diagrams. Hence disjunctive merge can be used for factoring out common features in diagrams. □

5 Semantic Soundness

In this section we give a precise semantics to class diagrams and use this to show that our constructions of subtyping $\leq$, composition $\owedge$, and difference $\oslash$

are semantically sound. That is, subtypings of a class diagram $\mathcal{C}$ have fewer implementations than $\mathcal{C}$, the implementations of a merge $\mathcal{C} \otimes \mathcal{D}$ are all also implementations of both $\mathcal{C}$ and $\mathcal{D}$, and the implementations of a difference $\mathcal{D} \oslash \mathcal{C}$ merged with $\mathcal{C}$ are also implementations of $\mathcal{D}$.

The semantics of a class diagram is given by a set of *instance diagrams*, or *object models*, which implement the class diagram. An instance diagram essentially consists of objects and links which are typed by classes and associations:

Definition 7. *An* instance diagram *is a tuple* $\mathcal{M} = (\mathsf{obj}, \mathsf{lnk}, \mathsf{oty}, \mathsf{lty}, \mathsf{lends})$ *of*

- *A finite set* obj *of* objects*,*
- *A finite set* lnk *of* links*,*
- *A total relation* $\mathsf{oty} \subseteq \mathsf{obj} \times \Sigma_c$ *associating objects with their* object types*,*
- *A total relation* $\mathsf{lty} \subseteq \mathsf{lnk} \times \Sigma_a$ *associating links with their* link types*,*
- *A partial function* $\mathsf{lends} : \mathsf{lnk} \to \Sigma_e \rightharpoonup \mathsf{obj}$ *mapping each link to its endpoints.*

We require that $|\mathsf{dom}(\mathsf{lends}(\ell))| = 2$ *for each* $\ell \in \mathsf{lnk}$.

We use the common notation "$a : A$" for object and link typing, instead of the more cumbersome "(a, A)". An object type relation specifying one object a of type A and another, b, of type B, will thus be denoted by $\mathsf{oty} = \{a : A, b : B\}$.

Definition 8. *An instance model* $\mathcal{M}$ *is said to* implement *a class diagram* $\mathcal{C}$*, denoted* $\mathcal{M} \models \mathcal{C}$*, if there exist extended typing relations* $\mathsf{Oty} \supseteq \mathsf{oty}$*,* $\mathsf{Lty} \supseteq \mathsf{lty}$ *such that the following hold:*

1. $|\{o \in \mathsf{obj} \mid \mathsf{Oty}(o, c)\}| \in \mathsf{ccard}(c)$ *for all* $c \in \mathsf{cla}$
(class cardinalities are respected)
2. *For all* $o \in \mathsf{obj}$ *and all* $c, c' \in \mathsf{cla}$ *with* $\mathsf{Oty}(o, c)$ *and* $\mathsf{gen}(c, c')$*, also* $\mathsf{Oty}(o, c')$
(object types are consistent with generalizations)
3. *For all* $\ell \in \mathsf{lnk}$ *and all* $a, a' \in \mathsf{asc}$ *with* $\mathsf{Lty}(\ell, a)$ *and* $\mathsf{gen}(a, a')$*, also* $\mathsf{Lty}(\ell, a')$
(link types are consistent with generalizations)
4. *For all* $\ell \in \mathsf{lnk}$ *and all* $a \in \mathsf{asc}$ *with* $\mathsf{Oty}(\ell, a)$*,* $\mathsf{dom}(\mathsf{lends}(\ell)) = \mathsf{dom}(\mathsf{aends}(a))$
(link endpoints inherit their names from their type)
5. *For all* $\ell \in \mathsf{lnk}$ *and all* $a \in \mathsf{asc}$ *with* $\mathsf{Lty}(\ell, a)$*, it holds for all* $e \in \mathsf{dom}(\mathsf{lends}(\ell))$ *that* $\mathsf{Oty}(\mathsf{lends}(\ell)(e), \mathsf{aends}(a)(e))$
(link endpoints are well-typed)
6. *For all* $o \in \mathsf{obj}$*,* $a \in \mathsf{asc}$ *and* $e \in \mathsf{dom}(\mathsf{aends}(a))$ *with* $\mathsf{Oty}(o, \mathsf{aends}(a)(e))$*, we have* $|\{\ell \in \mathsf{lnk} \mid \mathsf{Lty}(\ell, a) \;\&\; \mathsf{lends}(\ell)(e) = o\}| \in \mathsf{acards}(a)(e)$
(association end cardinalities are respected)
7. *There are no* $o \in \mathsf{obj}$*,* $c, c' \in \mathsf{cla}$ *for which* $\mathsf{disj}(c, c')$*,* $\mathsf{Oty}(o, c)$ *and* $\mathsf{Oty}(o, c')$
(disjointness constraints are respected)

The set of implementations *of* $\mathcal{C}$ *is* $[\![\mathcal{C}]\!] = \{\mathcal{M} \mid \mathcal{M} \models \mathcal{C}\}$.

The first, and most important, theorem in this section shows that subtyping is semantically sound. The proof is basically a careful inspection of the conditions for subtyping and implementation.

Theorem 6. *For all class diagrams* $\mathcal{C}_1$*,* $\mathcal{C}_2$*,* $\mathcal{C}_1 \leq \mathcal{C}_2$ *implies* $[\![\mathcal{C}_1]\!] \subseteq [\![\mathcal{C}_2]\!]$.

We sum up the semantic properties of our operators; all follow easily from their syntactic properties and Theorem 6. Note that $\owedge$ and $\oslash$ are semantically *sound*, i.e., under-approximations, whereas $\ovee$ is semantically *complete*.

Theorem 7. *For all pairs of class diagrams $\mathcal{C}_1$, $\mathcal{C}_2$: $[\![\mathcal{C}_1 \owedge \mathcal{C}_2]\!] \subseteq [\![\mathcal{C}_1]\!] \cap [\![\mathcal{C}_2]\!]$, $[\![\mathcal{C}_1 \owedge (\mathcal{C}_2 \oslash \mathcal{C}_1)]\!] \subseteq [\![\mathcal{C}_2]\!]$, and $[\![\mathcal{C}_1 \ovee \mathcal{C}_2]\!] \supseteq [\![\mathcal{C}_1]\!] \cup [\![\mathcal{C}_2]\!]$*

Intuitively, Thm. 7 states that our operators behave as expected *not only* with respect to the syntactic subtyping, as shown in Sections 3–4, but also with respect to the semantics (our ultimate goal). Crucially, Thm. 7 follows almost directly from the theorems of those sections, once we have Thm. 6. So the main work required to transfer the results of this paper to another semantic variation for class diagrams, is to obtain the equivalent of Thm. 6 for the new semantics.

Using simple examples, we can show that subtyping and composition are not semantically complete. For subtyping, we use inconsistency: Let $\mathcal{C}_1 = \langle A^{\emptyset} \rangle$ and $\mathcal{C}_2 = \langle B^{\emptyset} \rangle$. Then $[\![\mathcal{C}_1]\!] = [\![\mathcal{C}_2]\!] = \emptyset$, but $\mathcal{C}_1 \not\leq \mathcal{C}_2$, because $\mathcal{C}_2$ contains a restricted class which is not in $\mathcal{C}_1$; for the same reason, $\mathcal{C}_2 \not\leq \mathcal{C}_1$.

This also exposes the fact that $\leq$ does not have a unique bottom element; there is no class diagram $\mathcal{B}$ such that $\mathcal{B} \leq \mathcal{C}$ for all class diagrams $\mathcal{C}$.

For incompleteness of composition, we observe that generalizations can implicitly force object typing in one diagram which is forbidden in another: Let

$$\mathcal{C}_1 = \langle A \relbar\joinrel\relbar\joinrel\rhd B \rangle, \qquad \mathcal{C}_2 = \langle B^1 \rangle,$$

then $\mathcal{C}_1 \owedge \mathcal{C}_2 = \langle A \relbar\joinrel\relbar\joinrel\rhd B^1 \rangle$. Now let $\mathcal{M} = \{a : A, b : B\}$, the instance model with one object a of type A and another, b, of type B. Then $\mathcal{M} \models \mathcal{C}_1$, as witnessed by the extended object typing $\mathsf{Oty}_1 = \{a : A, a : B, b : B\}$, and $\mathcal{M} \models \mathcal{C}_2$, witness by $\mathsf{Oty}_2 = \mathsf{oty}_{\mathcal{M}} = \{a : A, b : B\}$. However, $\mathcal{M} \not\models \mathcal{C}_1 \owedge \mathcal{C}_2$, as any witness to this would have to include the typing $\{a : A, a : B, b : B\}$ with *two* objects of type B.

6 Conclusion and Final Remarks

We have presented a compositional algebra of class diagrams with subtyping, conjunctive and disjunctive merge and difference. All operators are described by means of manipulations of minimal syntactic elements of the diagrams, which are also basic terms of our class diagram algebra. To the best of our knowledge, this is the first attempt to define these syntactic operations in a provably sound manner. The operations are all efficiently computable and thus can be automated. The results of operations are represented in the syntax of the input language (they are class diagrams themselves), so that they can later be further processed using regular tooling for class diagrams.

We have worked with a simple core part of the class diagram language, which has allowed us to include all the technical constructions in the paper. Some extensions to other language elements are straightforward, some others require more extensive work. From the point of view of differencing, treating attributes is relatively simple, e.g. by boxing these as classes and treat them in the same way as classes. This is somewhat unsatisfactory though, as treating attributes specially would allow presenting the differences in a more concise and clear manner. Operations would require also computing differences between their signatures—we have decided not to discuss this, as it is not specific to structural modeling, but can be done using techniques for textual programming languages. To handle n-ary associations, one could introduce association classes and treat them in the same way as we treat regular classes. Allowing association classes to be evolved into regular classes (and vice-versa) would require more extensive changes though. Handling abstract classes is not much different from handling concrete ones, and the same goes for directed associations (vs. undirected ones). However, computing differences between diagrams where classes are changed from concrete to abstract, or associations from directed to undirected, or where association endpoints are moved, seems more challenging. Finally, note that we have used element identities to match them for the purpose of merging and differencing. Clearly, in a real application one should use an externally computed mapping, using for instance existing matching heuristics.

We have chosen to work with sound approximations for simplicity and clarity. Experience with building theories for behavioral models shows that a search for precise (i.e., both sound and complete) refinements and compositions leads to complex constructions with high computational complexity [4, 5], so it should only be done once the overall structure and properties of the design space are well understood on simpler cases. We speculate that more precise operators could be expressed in our algebra if we had a *complementation* for class diagrams. Unfortunately, in our current framework, complements of class diagrams will need infinitely many classes, hence are outside the syntax. Alternatively, to overcome the limitations of the class diagram notation, one can consider using Object Constraint Language to specify more precise differences and merges. We intend to investigate these possibilities in future work.

We remark that the semantics we give to class diagrams in this paper is only one out of a plethora of different existing class diagram semantics which are being used, c.f. [18]. We have shown that our constructions are semantically sound for our particular semantics, but this soundness may break if other semantics are used. However, to check that the constructions are sound, one only needs to see that subtyping is semantically sound, i.e. that the semantics is monotonic with respect to subtyping; if this is in place, then all other semantic properties follow. Hence our work lends itself easily to different semantics configurations [18], a point we intend to elaborate in the future. The work reported here is part of a larger project on model management. Our long term objective is to develop semantically sound (and reasonably complete) model management operations for other formalisms, beyond class diagrams and feature models [1, 2, 10].

References

1. M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, and R. B. France. Composing your compositions of variability models. In *MoDELS'13*, vol. 8107 of *LNCS*, pp. 352–369, 2013.

2. M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. Feature model differences. In *CAiSE'12*, vol. 7328 of *LNCS*, pp. 629–645, 2012.
3. M. Alanen and I. Porres. Difference and union of models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML*, vol. 2863 of *LNCS*, pp. 2–17. Springer, 2003.
4. A. Antonik, M. Huth, K. G. Larsen, U. Nyman, and A. Wąsowski. EXPTIME-complete decision problems for modal and mixed specifications. *Electr. Notes Theor. Comput. Sci.*, 242(1):19–33, 2009.
5. N. Beneš, J. Křetínský, K. G. Larsen, and J. Srba. EXPTIME-completeness of thorough refinement on modal transition systems. *Inf. Comput.*, 218:54–68, 2012.
6. Bibliography on comparison and versioning of software models. http://pi.informatik.uni-siegen.de/CVSM.
7. G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *GaMMa'06*, pp. 5–12. ACM, 2006.
8. G. Brunet, M. Chechik, and S. Uchitel. Properties of behavioural model merging. In *FM'06*, vol. 4085 of *LNCS*. Springer, 2006.
9. J. Euzenat and P. Shvaiko. *Ontology matching*. Springer, 2007.
10. U. Fahrenberg, A. Legay, and A. Wąsowski. Vision paper: Make a difference! (semantically). In Whittle et al. [24], pp. 490–500.
11. R. B. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing support for model composition in metamodels. In *EDOC*, pp. 253–266. IEEE, 2007.
12. J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *ICSE'11*. ACM, 2011.
13. D. Kolovos, R. Paige, and F. Polack. Merging models with the epsilon merging language (EML). In *MODELS'06*, vol. 4199 of *LNCS*. Springer, 2006.
14. D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *CVSM/ICSE*, pp. 1–6. IEEE, 2009.
15. S. Maoz, J. O. Ringert, and B. Rumpe. ADDiff: semantic differencing for activity diagrams. In *ESEC/FSE*, pp. 179–189. ACM, 2011.
16. S. Maoz, J. O. Ringert, and B. Rumpe. CDDiff: semantic differencing for class diagrams. In *ECOOP*, pp. 230–254. Springer, 2011.
17. S. Maoz, J. O. Ringert, and B. Rumpe. A manifesto for semantic model differencing. In *MODELS*, pp. 194–203. Springer, 2011.
18. S. Maoz, J. O. Ringert, and B. Rumpe. Semantically configurable consistency analysis for class and object diagrams. In Whittle et al. [24], pp. 153–167.
19. S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *ICSE*, pp. 54–64. IEEE, 2007.
20. Y. Reddy, S. Ghosh, R. France, G. Straw, J. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. In A. Rashid and M. Aksit, editors, *Trans. AOSD*, vol. 3880 of *LNCS*. 2006.
21. J. Rubin and M. Chechik. Combining related products into product lines. In J. Lara and A. Zisman, editors, *FASE*, vol. 7212 of *LNCS*, pp. 285–300. Springer, 2012.
22. J. Rubin and M. Chechik. Quality of merge-refactorings for product lines. In V. Cortellessa and D. Varró, editors, *FASE*, vol. 7793 of *LNCS*. Springer, 2013.
23. M. Sabetzadeh, S. Nejati, S. M. Easterbrook, and M. Chechik. Global consistency checking of distributed models with TReMer+. In *ICSE*, pp. 815–818, 2008.
24. J. Whittle, T. Clark, and T. Kühne, editors. *Model Driven Engineering Languages and Systems, MODELS 2011. Proceedings*, vol. 6981 of *LNCS*. Springer, 2011.
25. Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *ASE*, pp. 54–65. ACM, 2005.