JOT
JOURNAL OF
OBJECT TECHNOLOGY

[LRSS23] A. Lindt, B. Rumpe, M. Stachon, S. Stüber:
CDMerge: Semantically Sound Merging of Class Diagrams
for Software Component Integration.
In: Journal of Object Technology,
Volume 22(2), pp. 2:1-14, Jul. 2023.

CDMerge: Semantically Sound Merging of Class Diagrams for Software Component Integration

Achim Lindt, Bernhard Rumpe, Max Stachon, and Sebastian Stüber
Software Engineering, RWTH Aachen University, Germany

ABSTRACT

Abstraction, refinement and (de-)composition are fundamental techniques for engineering large software systems. In the context of Model Driven Development (MDD), these techniques are primarily applied to models. Our goal is to integrate automated composition of data models into the development process. We focus primarily on Class Diagrams (CDs) which are widely used to model object-oriented systems. In particular, we consider the variant of UML/P CDs which are equipped with a formal semantics for both the closed-world and open-world assumptions and allow for underspecified associations. A sound merge of CDs must include precisely the information of its components and preserve their semantic implications. In this paper we introduce a merge operator for CDs that considers both formal and implementation-oriented soundness requirements. The operator is able to support a divide-and-conquer approach for modeling and code-generation of large object-oriented software systems. We clarify why we deem an open-world approach necessary and outline merge conflicts and variants. Finally, we discuss integration of automated merging into the development process and provide an outlook on run-time data integration.

KEYWORDS class diagrams, uml, composition, merge, semantics, model-driven engineering.

1. Introduction

One paramount principle of software engineering is to achieve separation of concerns across software components. Consequently, larger software systems are usually composed of dedicated sub-modules. Ideally, there is loose coupling between and strong cohesion within components to achieve a high degree of adaptability and re-usability. Components should therefore be defined and integrated using interfaces that hide the internal behaviour and complexity. This then allows for development in separate teams with distinct responsibilities for certain modules (Nagl 1990).

In MDD, models become first-class programming artifacts and input to code generators (France and Rumpe 2007). But although there have been previous approaches to sound model composition (Boronat et al. 2007; Fahrenberg et al. 2014) and aspect-oriented modeling (Wimmer et al. 2011), integration of automated data model composition into the development process is still uncommon. The current situation leads to either of the two following pitfalls:

– Consistent but monolithic data models. These models mostly entail all data types that are required to generate the code for the complete target system and become inflated and incomprehensible, thus contradicting one of the major principle of MDD: reducing complexity via abstraction.
– Decomposed, but potentially inconsistent data models. Here each data model is used to characterize a specific component of the software system. However, such models will inevitably share and possibly extend common data types that are used across the system. Any conflicting model statement in such a datatype will then lead to inconsistency and errors.

Clearly, the first option contradicts the software engineering principles and is hence not favorable. To achieve the desired decomposed model architecture, a sound model composition

JOT reference format:
Achim Lindt, Bernhard Rumpe, Max Stachon, and Sebastian Stüber.
CDMerge: Semantically Sound Merging of Class Diagrams for Software Component Integration. Journal of Object Technology. Vol. 22, No. 2, 2023.

http://dx.doi.org/10.5381/jot.2023.22.2.a1

semantics and appropriate tool support is required to mitigate inconsistency. As generated code usually requires additional handwritten extensions (Greifenberg et al. 2015; Wile 2003), such composition semantics need to ensure two aspects: (1) that the semantic of the aggregated model is sound with respect to the semantics of each input model and (2) does not violate any implications of implementation in the handwritten or generated code. The component models could be interpreted as views of the integrated system in traditional data modeling. However, we assume that the component models remain the primary development artifacts and continue to have their own development life-cycle. As such, the aggregated model is merely virtual and created on demand to prove consistency among the component models and to generate (glue) code. In that sense, traditional views are top-down and originate from an integrated data model whereas composition of component models is bottom-up.

We focus on composing/merging UML/P CDs, a variant of UML CDs designed for agile development of object-oriented systems in Java (Rumpe 2011), as corresponding tooling for code-generation as well as a methodology for extending generated code-artifacts with handwritten code already exists (Schindler 2012).

In general, we seek to answer the question on how to (de-)compose the architecture of model driven systems, in particular its complex data structures, in order to support independent, generator-supported development of subsystems. As part of our contributions we address the following research questions:

1) How can a merge-operator for data models (*e.g.,* CDs) be defined such that it preserves semantic implications as well as consistency even for handwritten extensions of functionality?

2) How well can semantics-preserving composition support integration and re-usability of model driven software components?

Our contributions include a formal specification and implementation of a semantically sound CD-merge operator that is suitable for model-based integration of software components. Moreover, we discuss potential merge conflicts and variants of the merge operator, that consider both formal and practical aspects such as backwards-compatibility with preexisting software components. The implementation of the operator, referred to as `CDMerge`, has been integrated into the `CD4Analysis` Tool developed at the Chair for Software Engineering at RWTH Aachen University. The project is publicly available on GitHub[1].

The remainder of this paper is structured as follows: The next section discusses related work. Section 3 introduces a motivating example. Section 4 defines abstract syntax and formal semantics of CDs. Section 5 characterizes the notion of a semantically sound merge, defines a corresponding merge operator, and discusses potential merge conflicts and variants. In section 6, we evaluate the implementation of the merge operator and discuss runtime implications when integrating software components. The final section presents conclusions drawn from our research and discusses potential future work.

[1] `https://github.com/MontiCore/cd4analysis`

2. Related Work

The need for software merging has been prevalent for quite some time and (Mens 2002) discusses the problem domain in general and in particular the problems of *optimistic version control* where software artifacts are edited separately and independently which requires a merge to integrate the different versions. The author clearly stresses the importance of semantic based merging, which is also the main objective of our contribution. However, we do not aim to generate a merged class diagram which is then used by all parties as new version. Instead, we assume that each input model remains a first-class development artifact.

Aspect-Oriented Modeling (AOM) approaches Model-Driven integration of software components by merging functional requirements models with aspect models that deal with crosscutting concerns and are mostly non-functional (Wimmer et al. 2011; Johannes & Aßmann 2010). In AOM, models and meta models either require defined interfaces, usually referred to as pointcuts or hooks, or run within a framework to allow composition of models. Invasive Software Composition introduced in (Aßmann 2003) allows injection of fragment components, (*e.g.,* classes, methods or package implementations) into dedicated hooks. Our approach instead aims to integrate components in a more holistic manner and attempts to guarantee backwards-compatibility with preexisting implementations of each component CD.

Integration of data models is a well-known subject matter that was already being discussed in the early days of database views and schema integration by superimposition (Batini et al. 1986). In the remainder, we instead focus on related work that deals with the composition of UML CDs.

(Dingel et al. 2008) discuss UML package merge and its application for modular construction of data models via extension. Package merge was introduced for UML 2.0 in order to define the UML meta model in a modular way and in particular to specify UML compliance levels in a hierarchical manner. It specifies the composition of a source CD and a target CD that extends the source CD. Elements within the composite CDs are matched (mostly by name) and then merged. Merging of elements is either done (1) recursively, (2) by union, or (3) by property-specific transformation. Merge by union is, for example, applied to cardinalities of associations and as such conflicts with the formal notion of semantic refinement (Maoz et al. 2011; Nachmann et al. 2022). Moreover, associations are matched across super-/sub-classes and the merging is property-specific in that the merged association always references the most specific types. This not only conflicts with refinement but also backwards-compatibility with preexisting software components.

In (Boronat et al. 2007) an automated approach for generic model merging using the QVT Relations language is presented. The paper focuses on the definition of a merge operator for CDs. Set logic is used to describe the CDs themselves, as well as

the merge. However, association directions and underspecified associations are not supported. Merging of non-equal cardinalities via intersection is permitted, thus potentially violating implied "write"-promises[2]. In (Kolovos et al. 2006) the authors introduce the Epsilon Merging Language and provide a rule set for merging CDs. However, here the provided rules only allow matching of syntactically identical elements, *e.g.,* named associations with identical cardinalities. The concept of soundness based on a formal semantics for CDs is discussed in neither of the two publications. (Meier et al. 2020) discusses multi-view software development and how to ensure consistency across views. The approach proposed in the paper utilize a projective, top-down perspective from a Single Underlying Model (SUM), which functions as a single point of truth that is to be respected in all components corresponding to with of the SUM. It ensures that all views are consistent according to the integrity of the SUM and the semantically sound projection mechanism.

In (Lutz et al. 2011), the authors study how humans approach the merging of UML CDs. They identified categories of activities that the participants engaged in during the merging process and formulated guidelines for developing interactive merge tools based on their findings. These findings showed a preference for simple solution and preservation of functionality. During the merge process new elements would be added in order to produce a valid and/or semantics-preserving solution. Moreover, elements identified to be redundant or unnecessary were removed and perceived design flaws eliminated or corrected. An automated approach for merging CDs would certainly less flexible, but it also would ensure formal soundness. Furthermore, necessary adjustments can simply be made to the component diagrams before merging.

In (Fahrenberg et al. 2014) a merge and difference operator for CDs was introduced that uses a formal open-world semantics of permitted object structures in order to characterize the soundness-property. The authors introduce the notion of syntactic refinement as a sufficient condition for semantic refinement. This notion is similar to the concept of CD-expansion introduced in (Nachmann et al. 2022). The operator does not support attributes, association directions or underspecified associations. Furthermore, implementation-specific aspects not covered by the formal notion of refinement, which are needed in order to achieve backwards-compatibility with preexisting software components, are not considered either (*e.g.,* preserving "write"-promises of cardinalities in addition to "read"-promises when merging CD [2]).

Overall, we find that most existing approaches only discuss the merge semantics in terms of valid data instances or conformance with meta models and do not consider the implications on operational interactions from software components that are implemented in accordance to prevailing component models.

3. Motivating Example

As a motivating example we consider the development of a university administration system. The system is divided into two components to support both teaching activities and the management of staff. Both of these components have specific demands that require domain expertise (*e.g.,* curricula, accounting procedures) for the system design and hence should be implemented by dedicated software teams as separate subsystems. A MDD approach is taken and since CDs are widely used to model the data architecture of object-oriented software systems, the data model of each of the modules is a CD. Each CD can be interpreted as a view on the university system at large. However, they are in fact source-models and not just virtual artifacts derived from a larger source-model. The CDs are kept rudimentary for the purpose of demonstration.

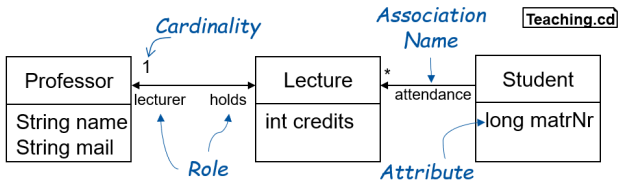

Figure 1 Teaching Subsystem: Planning of lectures and student registration.

The teaching subsystem is depicted in fig. 1. A `Lecture` is held by exactly one `lecturer` of type `Professor`. Via the `attendance` association a `Student` can register to multiple lectures. It is not specified how many lectures a professor holds, or how many students can attend a lecture. All classes have attributes relevant for the teaching subsystem, for example the `mail` address to contact a professor.

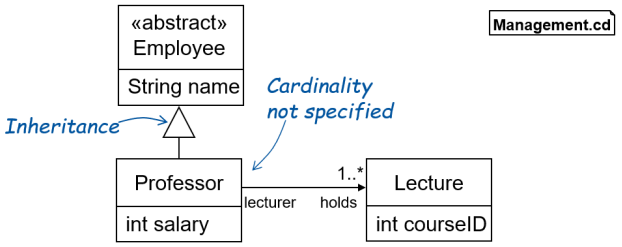

Figure 2 Management Subsystem

The data model of the management subsystem is shown in fig. 2. It has an abstract super-class `Employee` with an attribute `name`. A `Professor` is an `Employee`. They must hold at least one `Lecture`.

From their respective CD each development team generates code in a general programming language, *e.g.,* Java. The generated code can contain database access, validators or even visualisations (Reiß 2016; Dalibor et al. 2020; Michael et al. 2022). The actual business logic is written manually, either by importing or extending the generated code.

In this paper we analyze under which conditions and how the two subsystems can be merged such that the generated code and handwritten business logic of each component can be reused for the merged system. Figure 3 depicts the idealized workflow. First, the two teams responsible for developing the `Teaching` and `Management` subsystems define corresponding

[2] see Merge Conflict 5.5 in section 5.3

data-schemata and write the required business logic. Then the CDs are merged into one CD representing the data-architecture of the overarching system, denoted as `University`. Generated code from this composed model serves to integrate the sub-components with their corresponding business logic. Hence, the hand-coded business logic of subsystems must also be compatible with the generated code of the merged system. Note that the composed model is created on demand and thus only exists as a virtual artifact and does not have an independent life-cycle.

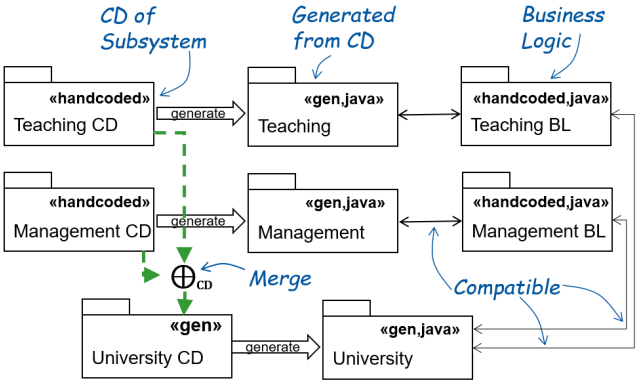

Figure 3 Distributed Model-Driven Engineering (MDE) using `CDMerge`. `Teaching` and `Management` is developed by separate teams in parallel. The handwritten business logic is compatible to the merged system.

A syntactical composition or merge of CDs should combine class declarations of the same name as well as attributes of the merged classes with the same name and datatype. Furthermore, associations are matched using role-names and association names. While it is straightforward to match named elements, this is not the case for associations that do not share an explicit name. The matching and merging of associations is further complicated by cardinalities on each side (or lack thereof) as well as their direction. Hence, in order to preserve the intended properties of the component CD, it is necessary to specify a formal semantics for CDs (see section 4) and then ensure that the merge semantically refines its components (see section 5). The merging of the two previously discussed example-subsystems results in the composite CD shown in fig. 4.

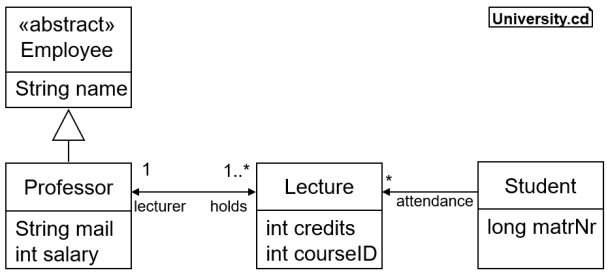

Figure 4 University System = Teaching $\oplus$ Management

For the implementation it is paramount, that each application either uses the data types of the composed model, or that there exists an infrastructure that introduces necessary data type adapters or wrappers to facilitate data exchange between modules. This might be necessary for security and privacy reasons, assuming that each module will result in an own application that is used by different entities in a university. For example, data instances of a `Professor` have an attribute `int salary` which should not be known to the Teaching Subsystem. These run-time integration aspects are discussed in section 6.2.

The takeaways of this motivating example are as follows

- A decomposed system architecture is favorable as each module addresses domain-specific requirements.
- Dedicated domain models for each module are more concise and comprehensive compared to a monolithic system model.
- It is necessary to merge all data models into a holistic model to create a fully integrated system.
- Type adapters may be required to handle run-time data exchange among the software components.

4. Class Diagrams and their Semantics

In this section, we give a formal specification of abstract syntax and semantics for CDs. Our notion of CDs is based on the UML/P variant as outlined in (Rumpe 2011). We consider a formal semantics that maps each CD to a set of valid objects structures (Harel & Rumpe 2004; Maoz et al. 2011; Nachmann et al. 2022). Furthermore, we distinguish between the closed-world semantics of a CD, which only includes instances of explicitly modelled elements, and the open-world semantics, that considers CDs as under-specified and thus permits instances of other, undeclared types and associations, as well. We have already discussed these notions of closed-world and open-world semantics in (Nachmann et al. 2022). However, since we now also consider association directions as well as underspecified associations, it is necessary to adjust our previous definitions.

For notation, let $\mathcal{C}$ be a universe of class names and $\mathcal{N}$ a universe of non-empty names for associations, roles and attributes. We also define $\mathcal{N}_\epsilon := \mathcal{N} \dot\cup \{\epsilon\}$. Moreover, let $\mathcal{T}$ be a set of basic predefined data types such as `int, bool, char, String` etc. Finally, for the purpose of defining association cardinalities, let $\mathcal{I}$ be the set of all finite unions of finite or unbounded intervals of natural numbers.

4.1. Abstract Syntax of Class Diagrams

A CD defines a finite set of classes, as well as a finite set of associations between those classes. Each class contains a finite set of attributes and each attribute consists of a name and a type. A class may extend other classes, thus inheriting all attributes and associations of its super-classes. Moreover, classes can be declared abstract. Abstract classes cannot be instantiated directly and may only extend other abstract classes. Associations can be either bidirectional, unidirectional or have no specified direction. An association may also have a unique name, as well as a role name and cardinality for each side. The cardinality imposes constraints on the number of allowed instances (*i.e.,* links) referencing the same object. Formally, we define the abstract syntax of a CD as follows:

Definition 4.1 (Class Diagram)**.** *Given a Class Diagram cd we declare that*

- $cd.classes \subseteq \mathcal{C} \backslash \mathcal{T}$*: denotes the finite set of class declarations in cd,*
- $cd.abstract \subseteq cd.classes$*: denotes the subset of classes declared abstract in cd,*
- $cd.attr \in cd.classes \to \wp_{fin}\{\mathcal{N} \times \mathcal{T}\}$*: is a function that assigns each class in cd a finite set of attributes,*
- $c_1 \prec_{cd} c_2$*: denotes that a class* c_1 *directly extends another class* c_2 *in cd,*
- $cd.assoc \subseteq cd.classes \times \mathcal{N}_\epsilon \times \mathcal{N}_\epsilon \times cd.classes$*: denotes the finite set of associations in cd,*
- $cd.nav \in cd.asssoc \to \wp\{\to, \leftarrow\}$*: is a function that assigns each associations in cd its directions,*
- $cd.label \in cd.assoc \rightharpoonup \mathcal{N}$*: is a partial function that assigns a unique name to an associations in cd,*
- $cd.cardL, cd.cardR \in cd.assoc \rightharpoonup \mathcal{I}$*: are two partial functions that respectively assign a cardinality to the left and right side of an association in cd*

Additionally, we require that the following context conditions hold:

1. *The reflexive transitive hull of the extends-relation* $\prec_{cd}$*, denoted by* $\prec^*_{cd}$*, must define a partial ordering on* $cd.classes$*.*
2. *The transitive attributes of a class* $c \in cd.classes$*, denoted by* $cd.attr^*(c) := \bigcup_{c \prec^*_{cd} s} cd.attr(s)$*, must be conflict free, i.e., for two attributes* $(n_1, t_1), (n_2, t_2) \in cd.attr^*(c)$ *it holds that* $n_1 = n_2 \implies t_1 = t_2$*.*
3. *There are no duplicate associations, i.e.,* $(c_1, r_1, r_2, c_2) \in cd.assoc \implies (c_2, r_2, r_1, c_1) \notin cd.assoc$*.*

Consider, for example, the CD in fig. 2. The set of class declarations is given by $cd.classes = \{Employee, Professor, Lecture\}$, the subset of abstract classes is $cd.abstract = \{Employee\}$ and the set of associations is $cd.assoc = \{(Professor, lecturer, holds, Lecture)\}$. $Professor$ extends $Employee$, denoted by $Professor \prec_M Employee$, and contains the attribute $(salary, int) \in cd.attr(Professor)$. The association $a := (Professor, lecturer, holds, Lecture)$ is unidirectional with $cd.nav(a) = \{\to\}$ and has a right cardinality of $cd.cardR(a) = [1, \infty)$.

It should be noted that the following elements of UML/P CDs were omitted from this definition: (1) method signatures, (2) composition, (3) interfaces, and (4) enumerations. These elements were excluded for two main reasons, first their inclusion in the formal syntax and semantics would exceed the scope of this paper, second their consideration with respect to the static semantic is superfluous. However, unlike (Nachmann et al. 2022) or (Fahrenberg et al. 2014), we do consider attributes and association directions, as well as underspecification for cardinalities, association directions and role names.

Next, we introduce the notion of a normalized CD. Normalization ensures that an associations orientation corresponds to lexicographic order. This is helpful when comparing associations of multiple CDs as we no longer have to check for the reverse notation, as well.

Definition 4.2 (CD-Normalization)**.** *Let* $\leq_{\mathcal{C}}$ *denote a relation that defines a lexicographic order on* $\mathcal{C}$*, and let* $\leq_{\mathcal{N}_\epsilon}$ *be analogous for* $\mathcal{N}_\epsilon$*. A Class Diagram cd is normalized iff for all associations* $(c_1, r_1, r_2, c_2) \in cd.assoc$ *it holds that* $c_1 \leq_{\mathcal{C}} c_2$ *and* $c_1 = c_2 \implies r_1 \leq_{\mathcal{N}_\epsilon} r_2$*.*

4.2. Semantic Domain

The semantic domain for CDs consists of object structures, which represent potential data states of an object-oriented system. An object structure consists of a finite set of objects, as well as a finite set of directed links. Each object instantiates a class and contains a set of attributes, and each attribute has a type and a name. Formally, we denote an object structures as follows:

Definition 4.3 (Object Structure)**.** *An object structure os consists of*

- *a finite set of objects* $os.obj$ *such that for each* $o \in os.obj$
  - $o.type \in \mathcal{C} \backslash \mathcal{T}$ *denotes its (most-specific) type,*
  - *and* $o.attr \subseteq \mathcal{N} \times \mathcal{T}$ *denotes its set of attributes,*
- *as well as a finite set of links between objects:* $os.links \subseteq os.obj \times \mathcal{N} \times os.obj$*.*

Note that we abstract from the concrete value of an attribute within an object structure, as attribute values are not relevant for the remainder of the paper.

4.3. Closed-World Semantics

First, let us consider the closed-world assumption. Informally, we may say that an object structure is a legal instance of a CD in the closed-world, iff the following conditions hold: (1) Each object within the object structure must instantiate a non-abstract class from the CD. (2) Each object must have precisely those attributes specified in the classes it instantiates. (3) Each link must correspond to an association. (4) The cardinality constraints of each association have to be respected.

To refer back to our motivating example in section 3, any instance of `Management.cd` (2) must contain only objects of type `Professor` or `Lecture` and each object of type `Professor` must be linked to at least one object of type `Lecture`.

For our formal definition of closed-world semantics, it is sufficient to consider fully-specified CDs. A CD is fully specified if all associations have non-empty role-names as well as specified directions and cardinalities. To further abbreviate the definition, we write $cd^{\to}$ to denote a modified Class Diagram cd, where each right-to-left association has been replaced with an equivalent left-to-right association.

Definition 4.4 (Closed-World Semantics of Class Diagrams)**.** *An object structure os is a closed-world instance of a fully specified Class Diagram cd iff the os satisfies the following constraints*

1. *For every object $o \in os.obj$, it holds that $o.type \in cd.classes \backslash cd.abstract$.*

2. *For every object $o \in os.obj$, it holds that $(n,t) \in o.attr \iff (n,t) \in cd.attr^*(o.type)$*

3. *For every link $(o_1, r_2, o_2) \in os.links$, there is an association $a \in cd^{\rightarrow}.assoc$ with $a = (t_1, r_1, r_2, t_2)$ such that $o_1.type \prec^*_{cd} t_1$ and $o_2.type \prec^*_{cd} t_2$.*

4. *For each $a := (c_1, r_1, r_2, c_2) \in cd^{\rightarrow}.assoc$ and each $o_2 \in os.obj$ with $o_2.type \prec^*_{cd} c_2$, it holds that $|\{(o_1, r_2, o_2) \in os.links : o_1.type \prec^*_{cd} c_1\}| \in cd^{\rightarrow}.cardL$*

5. *For each $a := (c_1, r_1, r_2, c_2) \in cd^{\rightarrow}.assoc$ and each $o_1 \in os.obj$ with $o_1.type \prec^*_{cd} c_1$, it holds that:*

   (i) $(o_1, r_2, o_2) \in os.links \implies o_2.type \prec^*_{cd} c_2$

   (ii) $|\{(o_1, r_2, o_2) \in os.links\}| \in cd^{\rightarrow}.cardR$

6. *For each $a = (c_1, r_1, r_2, c_2) \in cd.assoc$ that is bidirectional, i.e., $cd.nav(a) = \{\leftarrow, \rightarrow\}$, and each $o_1, o_2 \in os.obj$ with $o_1.type \prec^*_{cd} c_1$ and $o_2.type \prec^*_{cd} c_2$, we have: $(o_1, r_2, o_2) \in os.links \iff (o_2, r_1, o_1) \in os.links$.*

The closed-world semantics $[\![cd]\!]^{cw}$ of cd is the set of all closed-world instances.

A semantic refinement constitutes a restriction of permitted object structures. As such, a CD A is a refinement of a CD B under the closed-world assumption iff $[\![A]\!]^{cw} \subseteq [\![B]\!]^{cw}$ Note that a refinement in the closed-world does not necessarily constitute a refinement in the open-world, and vice-versa. (Nachmann et al. 2022)

4.4. Open-World Semantics

In order to accurately characterize open-world semantics for CDs, we employ the notion of CD-expansions introduced in (Nachmann et al. 2022). Informally, a CD-expansion must preserve preexisting classes, associations and extends relation, but additions are permitted. Associations may also be refined by reducing underspecification.

Definition 4.5 (CD-Expansion)**.** *A Class Diagram cd^+ is an expansion of a normalized Class Diagram cd iff*

1. *$cd.classes \subseteq cd^+.classes$,*

2. *$cd.abstract \subseteq cd^+.abstract$,*

3. *$\forall c_1, c_2 \in cd.classes : c_1 \prec_{cd} c_2 \implies c_1 \prec_{cd^+} c_2$,*

4. *$\forall c \in cd.classes : cd.attr^*(c) \subseteq cd^*attr^*(c)$,*

5. *and for each association $a = (c_1, r_1, r_2, c_2) \in cd.assoc$ there is a (syntactic) refinement in $cd^+.assoc$, i.e., there must be an association $a' = (c_1, r_1', r_2', c_2) \in cd^+.assoc$ with*

   - *$r_1 \neq \epsilon \implies r_1' = r_1$*
   - *$r_2 \neq \epsilon \implies r_2' = r_2$*
   - *$cd.nav(a) \subseteq cd^+.nav(a')$*
   - *$cd^+.label(a') = cd.label(a)$ if defined.*
   - *$cd^+.minL(a') = cd.minL(a)$ if defined.*
   - *$cd^+.maxL(a') = cd.maxL(a)$ if defined.*
   - *$cd^+.minR(a') = cd.minR(a)$ if defined.*
   - *$cd^+.maxR(a') = cd.maxR(a)$ if defined.*

   Moreover, we say that a Class Diagram cd^+ is an expansion of a set of Class Diagrams S iff cd^+ is an expansion of each Class Diagram $s \in S$.

In our running example (fig. 4), `University.cd` is an expansion of both `Teaching.cd` (fig. 1) and `Management.cd` (fig. 2) as it contains all elements from both CDs.

(Fahrenberg et al. 2014) employed a similar concept under the name "syntactic refinement" to characterize a merge. However the authors did not account for underspecified association and association directions. Moreover, we do not permit restriction of defined cardinalities. Although it constitutes a refinement with respect to permitted object structures, a restriction of an association cardinality may still correspond to a breaking change in the implementation (see section 5.3, conflict 5.5).

Definition 4.6 (Open-World Semantics of Class Diagrams)**.** *An object structure os is an open-world instance of a Class Diagram cd iff there exists a fully specified expansion cd^+ of cd with $os \in [\![cd^+]\!]^{cw}$. The open-world semantics of cd, denoted as $[\![cd]\!]^{ow}$, is the set of all open-world instances.*

In contrast to the closed-world semantics from definition 4.4, this notion of open-world semantics is well defined even for CDs that are not fully-specified themselves, as each of them has a fully-specified expansion (simply add specification to underspecified associations).

We say that a CD A is a refinement of a CD B under the the open-world assumption iff $[\![A]\!]^{ow} \subseteq [\![B]\!]^{ow}$ (Nachmann et al. 2022).

5. Sound Merging for Class Diagrams

To address research question 1, we specify the formal aspects of merging CDs and define a merge operator that is sound with respect to the formal semantics of CDs as outlined in section 4. Moreover, we outline the potential conflicts that may occur when merging two incompatible CDs, as well as discuss variants of the merge operator to handle these merge conflicts.

5.1. A Semantically Sound Merge

A sound merge of CDs must itself be a syntactically valid CD that contains precisely the information present in its components *i.e.,* it does not include any superfluous elements not found in its components but preserves their semantic implications. With regards to the formal semantics, this means that a merge must refine its component-CD. For this purpose, the closed-world assumption is unsuited, as a proper merge needs to include precisely those elements present in its components. This, however, would normally not constitute a closed-world refinement, except in trivial cases. Consider for example the CD`University.cd` in fig. 4, it is not a refinement of the CD`Management.cd` (fig. 2) in the closed-world as it permits

objects of type `Student`, which are not included in the closed-world semantics of `Management.cd`.

We therefore define a semantically sound merge of CDs as the minimal expansion of all component-CDs. This ensures that the merge contains only the elements/information already present in its component-CDs and also refines each of them under the open-world assumption.

Definition 5.1 (Semantically Sound Merge of Class Diagrams)**.** *Given a Class Diagram $cd_\oplus$, as well as a finite set of Class Diagrams S, we say that $cd_\oplus$ is a semantically sound merge of all $cd \in S$ iff $cd_\oplus$ is a minimal expansion of S, i.e., there is no expansion cd' of S such that $cd_\oplus$ is an expansion of cd' but cd' is not also an expansion of $cd_\oplus$.*

In our motivating example, `University.cd` (fig. 4) fulfils these requirements with regards to `Teaching.cd` (1) and `Management.cd` (fig. 2) as it does not contain any elements or specifications that are not also present in at least one of its components. Now it remains to show that the semantic implication of each component-CD are preserved by the merge.

Theorem 5.2. *Given a set of Class Diagrams S, let be cd be a semantically sound merge of S. For each $s \in S$, it holds that $[\![cd]\!]^{ow} \subseteq [\![s]\!]^{ow}$, i.e., cd is a refinement of s in the open-world.*

Proof. By definition 5.1 cd is an expansion of each $s \in S$. Furthermore, any expansion cd^+ of cd is also an expansion of s. It follows then that every open-world instance of cd is an open-world instance of s, as well. □

It should be noted that not every set of CDs possesses a semantically sound merge, as two different component-CDs A and B may contain conflicting definitions of the same class or association (see section 5.3).

5.2. Defining a Merge Operator for Class Diagrams

Let $\mathcal{CD}$ be the set of all syntactically valid CDs. In the following we define the merge operator $\oplus : \mathcal{CD} \times \mathcal{CD} \rightharpoonup \mathcal{CD}$ that given two input-CDs with a common expansion and unambiguous matching for associations produces a semantically sound merge.

In order to merge two CDs, we first have to identify and then merge matching classes and associations and then include the remaining elements of both CDs. Classes are matched by name and merging them requires merging their sets of attributes, as well as the extends relations. The matching associations is a bit more involved, as they do not always posses a unique association name.

Definition 5.3 (Matching Associations)**.** *Without loss of generality, let cd_1, cd_2 be two normalized Class Diagrams. Two associations $a_1 = (c_{11}, r_{11}, r_{12}, c_{12}) \in cd_1.assoc$, $a_2 = (c_{21}, r_{21}, r_{22}, c_{22}) \in cd_2.assoc$, are matching candidates for each other iff the following conditions hold*

1. *Classes match:*
   $c_{11} = c_{21}$ *and* $c_{12} = c_{22}$.
2. *Explicit match of association name if defined:*
   $cd_1.label(a_1) \neq \bot \neq cd_2.label(a_2)$
   $\implies cd_1.label(a_1) = cd_2.label(a_2)$
3. *Roles match if non-empty:*
   (i) $r_{11} \neq \epsilon \neq r_{21} \implies r_{11} = r_{21}$
   (ii) $r_{12} \neq \epsilon \neq r_{22} \implies r_{12} = r_{22}$

These two matching candidates are then matched iff (1) there is no other matching candidate $a_1' \in cd_1.assoc$ for a_2 or $a_2' \in cd_2.assoc$ for a_2, and (2) their cardinalities match if defined, i.e., $cd_1.cardL(a_1) = cd_2.cardL(a_2)$ and $cd_1.cardR(a_1) = cd_2.cardR(a_2)$ respectively. We then write $a_1 \sim_M a_2$ to denote that a_1 matches a_2. If all matching candidates are matched, we say that cd_1 and cd_2 have a non-ambiguous matching of associations.

Two matching associations are equal in their classes, rolenames, direction and cardinalities if specified. This makes the merge straightforward.

Definition 5.4 (Merging Associations)**.** *Without loss of generality, let cd_1, cd_2 be two normalized Class Diagram. For each pair of matching associations (a_1, a_2) with $a_1 = (c_1, r_{11}, r_{12}, c_{12}) \in cd_1.assoc$ and $a_2 = (c_2, r_{21}, r_{22}, c_2) \in cd_2.assoc$, the merged association $a_\oplus = (c_1, r_{1\oplus}, r_{2\oplus}, c_2)$ has the following properties:*

1. $r_{1\oplus} := \begin{cases} r_{11} & \text{, if } r_{11} \neq \epsilon, \\ r_{21} & \text{, if } r_{21} \neq \epsilon, \\ \epsilon & \text{, otherwise.} \end{cases}$
2. $r_{2\oplus} := \begin{cases} r_{12} & \text{, if } r_{12} \neq \epsilon, \\ r_{22} & \text{, if } r_{22} \neq \epsilon, \\ \epsilon & \text{, otherwise.} \end{cases}$
3. $cd_\oplus.nav(a_\oplus) = cd_1.nav(a_1) \cup cd_2.nav(a_2)$
4. $cd_\oplus.label(a_\oplus) := \begin{cases} cd_1.label(a_1) & \text{, if defined,} \\ cd_2.label(a_2) & \text{, if defined,} \\ \bot & \text{, otherwise.} \end{cases}$
5. $cd_\oplus.cardL(a_\oplus) := \begin{cases} cd_1.cardL(a_1) & \text{, if defined,} \\ cd_2.cardL(a_2) & \text{, if defined,} \\ \bot & \text{, otherwise.} \end{cases}$
6. $cd_\oplus.cardR(a_\oplus) := \begin{cases} cd_1.cardR(a_1) & \text{, if defined,} \\ cd_2.cardR(a_2) & \text{, if defined,} \\ \bot & \text{, otherwise.} \end{cases}$

The set of merged associations for cd_1 and cd_2 is then denoted as $assoc_\oplus(cd_1, cd_2)$.

Note that we do not merge associations between superclasses with associations between sub-classes, *i.e.,* we don't consider a "pull up" of associations for the merge as this can lead to conflicts in software component implementations (see 5.3, conflict 5.7).

Definition 5.5 (Merge Operator for Class Diagrams)**.** *The binary merge operator for two Class Diagrams is formally defined*

as the partial function $\oplus : \mathcal{CD} \times \mathcal{CD} \rightharpoonup \mathcal{CD}$ *that given two Class Diagrams* cd_1 *and* cd_2 *as inputs, produces a Class Diagram* $cd_\oplus := cd_1 \oplus cd_2$ *iff* cd_1 *and* cd_2 *have a non-ambiguous matching of association and the following specifications are syntactically valid according to definition 4.1:*

1. $cd_\oplus.classes := cd_1.classes \cup cd_2.classes$
2. $cd_\oplus.abstract := cd_1.abstract \cup cd_2.abstract$
3. $\forall c \in cd_\oplus.classes : cd_\oplus.attr(c) := cd_1.attr(c) \cup cd_2.attr(c)$
4. $\forall c_1, c_2 \in cd_\oplus.classes :$
   $c_1 \prec_{cd_\oplus} c_2 \iff c_1 \prec_{cd_1} c_2 \vee c_1 \prec_{cd_2} c_2$
5. $cd_\oplus.assoc := assoc_\oplus(cd_1, cd_2)$
   $\dot\cup\{a_1 \in cd_1.assoc \mid \nexists a_2 \in cd_2.assoc : a_1 \sim_M a_2\}$
   $\dot\cup\{a_2 \in cd_2.assoc \mid \nexists a_1 \in cd_1.assoc : a_2 \sim_M a_1\}$

For the most part, the merge operator simply unifies corresponding element-sets. Additionally, each pair of matching associations is replaced with a corresponding merged association. Although the resulting CD may contain redundant attribute definitions and class-extensions, these are not considered syntactical errors according to definition 4.1 nor do they affect minimality of CD-expansion. Note that in our implementation these redundancies are removed.

Theorem 5.6. *Given two Class Diagrams* cd_1 *and* cd_2 *with a common expansion and a non-ambiguous matching for associations, it holds that* $cd_1 \oplus cd_2$ *is a semantically sound merge.*

Proof. If $cd_1 \oplus cd_2$ is undefined, then cd_1 and cd_2 must either contain a conflicting attribute declaration, *i.e.,* there is a class $c \in cd_\oplus$ with attributes $(n, t_1), (n, t_2) \in cd_\oplus.attr^*(c)$ such that $t_1 \neq t_2$, or conflicting extends relations such that there exists two classes $c_1, c_2 \in cd_\oplus$ with $c_1 \prec^*_{cd_2} c_2$ and $c_2 \prec^*_{cd_1} c_1$. It follows then that there can be no common expansion of cd_1 and cd_2.

If $cd_1 \oplus cd_2$ is defined, then $cd_\oplus$ preserves all classes, attributes and the transitive extends relation as well as refines each associations of both cd_1 and cd_2. Consequently, $cd_\oplus$ is a common expansion of cd_1 and cd_2.

Now, let cd' be a common expansion of cd_1 and cd_2 such that $cd_\oplus$ is an expansion of cd'. Assume that cd' is not an expansion of $cd_\oplus$, then at least one of the following conditions has to hold:

1. $cd'.classes \subsetneq cd_\oplus.classes$
2. $cd'.abstract \subsetneq cd_\oplus.abstract$
3. $\exists c \in cd'.classes : cd'.attr(c)^* \subsetneq cd_\oplus.attr(c)^*$
4. $\exists c_1, c_2 \in cd'.classes : c_1 \prec_{cd_\oplus} c_2$
5. $\exists a \in cd_\oplus.assoc : a$ is not refined by any $a' \in cd'.assoc$

Since the merge operator $\oplus$ does not add any unnecessary elements or refinements of associations, it follows that cd' is not a common expansion of cd_1 and cd_2. We conclude that $cd_\oplus$ is a minimal expansion of $\{cd_1, cd_2\}$ and thus a semantically sound merge. □

In addition to formal soundness, we can identify the following properties of the merge operator: Given 3 Class Diagrams cd_1, cd_2, cd_3, the merge operator is

1. conditionally idempotent, *i.e.,* $cd_1 \oplus cd_1 = cd_1$ holds iff cd_1 has a non-ambiguous matching of associations,
2. commutative, *i.e.,* $cd_1 \oplus cd_2 = cd_2 \oplus cd_1$,
3. conditionally associative, *i.e.,* $(cd_1 \oplus cd_2) \oplus cd_3 = cd_1 \oplus (cd_2 \oplus cd_3)$ holds if all associations have a defined association name or if all associations have two non-empty role names.

5.3. Merge Conflicts

As mentioned before, not all pairs of CDs have a common expansion or a non-ambiguous matching of associations. Consequently, the merge operation may not always produce a valid CD. Conflicts arise from ambiguity and under-specification within the component diagrams. Some are detected during the merge process, others by checking the model integrity of the merge result. Variants of the merge operation can resolve certain conflicts at the cost of formal soundness.

Merge Conflict 5.1 (Attribute Type Violation)**.** *An attribute type violation occurs if a class c contains or inherits two attributes with the same name but different types, i.e.,* $(n, t_1) \in cd_1.attr^*(c)$ *and* $(n, t_2) \in cd_2.attr^*(c)$ *with* $t_1 \neq t_2$*. As those attributes do not match they cannot be merged into one target attribute.*
Result: *Error*
Variant: *Apply implicit type conversion.*

Figure 5 illustrates the attribute type conflict for the attribute `birthdate` which is either of type `long` in Unix format or as `String`. The proposed variant of implicit type conversion for base data types, as it is applied by general purpose programming languages such as Java, would merge two attributes (n, int) and $(n, long)$ into $(n, long)$. This resolves the conflict, however, there is now a danger of buffer overflows for any software component that is implemented according to this type specification, since `int` variables cannot handle `long` values that exceed the range of integer.

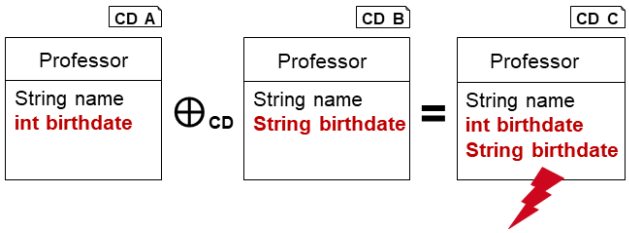

Figure 5 Merge Conflict: Attribute with same name but incompatible type

Merge Conflict 5.2 (Abstract Class)**.** *A class c is declared* abstract *in the first Class Diagram* cd_1 *whereas the other Class*

Diagram cd_2 declares c as non-abstract, or vice-versa.
Result: *Warning, the merged class will be declared abstract.*
Variants: *(1) Warning, the merged class will be declared non-abstract. (2) Error*

Declaring the merged class abstract will ensure formal soundness of the composite CD. If the merged class was declared non-abstract instead, the semantics of the composite CD would permit direct instances of the class. This would contradict the formal notion of refinement. On the other hand, any valid implementation for the component diagram cd_2 that creates direct instances of the class c would be incompatible with the resulting merge. As such, it might be warranted to (1) sacrifice formal soundness in favor of backwards-compatibility with the component implementation in certain cases or alternatively (2) prohibit merging of abstract and non-abstract classes.

Merge Conflict 5.3 (Type Hierarchy Violation)**.** *A sound merge must preserve the type hierarchies $\prec^*_{cd_1}$ and $\prec_{cd_2}$ of its components cd_1 and cd_2. Assume that for classes $A, B \in cd_1.classes \cap cd_2.classes$ we find $A \prec^*_{cd_1} B$ and $B \prec^*_{cd_2} A$, i.e., A is a subclass of B in cd_1 while B is a subclass of A in cd_2. The merge operator would now produce an inheritance cycle $A \prec^*_{cd_\oplus} B \prec^*_{cd_\oplus} A$ which violates the partial order property of the transitive inheritance relation.*
Result: *Error*

Merge Conflict 5.4 (Association Conflicts with Attribute)**.** *Given components cd_1 and cd_2 with a common class $c_1 \in cd_1.classes \cap cd_2.classes$, as well as an attribute $(n,t) \in cd_1.attr(c_1)$ and an association $a \in cd_2.assoc = (c_1, r_1, n, c_2)$. The expression $cd_\oplus.n$ is ambiguous as it denotes either the attribute of the class or an instance of c_2 that is accessed by the same role name.*
Result: *Error*
Variant: *This ambiguity can be tolerated if the navigation direction is explicitly $cd.nav(a) \in \{\leftarrow\}$ as this prevents direct access to object instances of c_2 via the role n. In that case $cd_1.n$ always denotes the attribute.*

Merge Conflict 5.5 (Association with Mismatching Cardinalities)**.** *Let $a_1 \in cd_1.assoc$ and $a_2 \in cd_2.assoc$ be two matching associations in the respective Class Diagram. Then $cd_1.cardL(a_1) \neq \bot \neq cd_2.cardL(a_2) \implies cd_1.cardL(a_1) = cd_2.cardL(a_2)$ and $cd_1.cardR(a_1) \neq \bot \neq cd_2.cardR(a_2) \implies cd_1.cardR(a_1) = cd_2.cardR(a_2)$*
Result: *Error*

Cardinalities on each side of the association will only be merged if they are identical or unspecified. This restriction is not necessary if we simply require a formal refinement. Consequently, previous approaches that specified a merge operator for CDs (Fahrenberg et al. 2014; Boronat et al. 2007) merge cardinalities via intersection. However, for implementation purposes, we have to not only consider implied "read"-promises but "write"-promises, as well. Consider, for example, the CDs

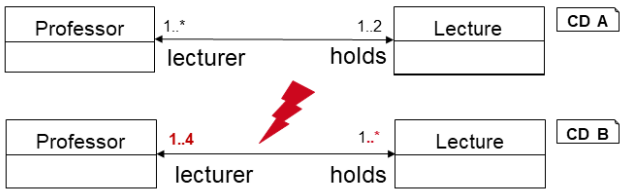

Figure 6 Merge Conflict: Mismatching Cardinalities

in fig. 6. Assume that we have a preexisting software component that corresponds to `CD A`. This software component might add arbitrarily many `Lecture` instances to `holds` for some instance of `Professor`. Now assume that we permit a merge of `CD A` with `CD B` that merges cardinalities via intersections. From the perspective of `CD A`, the cardinality for `holds` is thereby decreased from $[1,\infty)$ to $[1,2]$. We now might run into backwards-compatibility issues with our preexisting software component, as the component might still add more than two `Lecture` instances to `holds` for some instance of `Professor`.

Merge Conflict 5.6 (Associations with Ambiguous Match)**.** *Let $a_1 = (c_1, r_1, r_2, c_2) \in cd_1.assoc$ and $a_2 = (c_1, r_1, \epsilon, c_2) \in cd_2.assoc$ and $a_3 = (c_1, \epsilon, r_2, c_2) \in cd_2.assoc$ be unnamed associations in their respective Class Diagrams. There are multiple matching candidates for association a_1 as either of its roles matches exactly one association role in the other Class Diagram.*
Result: *Warning, Preserve all associations*
Variant: *Error*

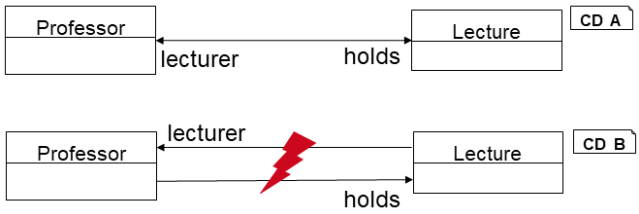

Figure 7 Merge Conflict: Ambiguous match of associations

Simply preserving all 3 associations in fig. 7 might produce a syntactically valid expansion, however, the semantics of the composite CD would effectively consider the intersection of all cardinalities and such an implicit change in these cardinality constraints would be problematic for the reasons outlined before.

Merge Conflict 5.7 (Ambiguous Roles)**.** *Let $a_1 = (c_1, r_1, r, c_2) \in cd_1.assoc$ and $a_2 = (c_1, r_2, r, c_3) \in cd_2.assoc$ and $c_1 \neq c_3$. The role r is ambiguous for the resulting association if $cd_1.nav(a_1) \in \{\rightarrow, \leftrightarrow\}$ or $cd_2.nav(a_2) \in \{\rightarrow, \leftrightarrow\}$ and creates a type conflict navigating from the class c_1 as $c_1.r$ refers either to type c_2 or to type c_3. The same conflict for right-to-left associations is defined analogous.*
Result: *Error*

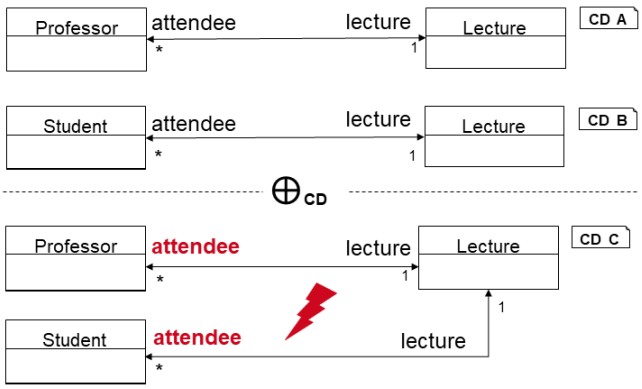

Figure 8 Merge Conflict: Ambiguous roles `Lecture.attendee`

This merge conflict is depicted in fig. 8: The role `Lecture.attendee` refers to ambiguous data types Professor vs. Student. Note that this remains a conflict even if, *e.g.*, c_3 is sub-class of c_2: Assume that an implementation for one component expects to navigate from an instance of the sub-class on the left side to an instance of the sub-class on the right side. If it finds an instance of the super-class (that was added by an implementation for the other component), it might run into an error when trying to access an operation or data that is only defined for the sub-class.

Finally, it should be mentioned that certain programming languages such as Java do not permit multiple-inheritance for classes. If this is the case for the implementation language, the following conflict has to be considered, as well:

Merge Conflict 5.8 (Multiple Inheritance)**.** *The merge will preserve class extensions declared in the component diagrams. Assume that component diagram cd_1 declares classes $A \prec_{cd_1} B$ whereas component diagram cd_2 declares $A \prec_{cd_2} C$. Merging both type hierarchies will result in multiple inheritance, i.e., $A \prec_{cd_\oplus} B$ and $A \prec_{cd_\oplus} C$ hold at the same time.*
Result: *Error*
Variant: *Interleave class extensions*

The proposed variant would interleave class extension such that $A \prec_{cd_\oplus} B \prec_{cd_\oplus} C$ or $A \prec_{cd_\oplus} C \prec_{cd_\oplus} B$ holds. This would only correspond to a weakened minimality with respect to CD-expansion where only CDs without multiple inheritance are considered. Additionally, it would weaken conditional associativity of the merge operator, as *e.g.*, $A \prec_{cd_\oplus} B \prec_{cd_\oplus} C$ might be chosen for $cd_1 \oplus cd_2$ and thus, given a third Class Diagram cd_3 that specifies $C \prec_{cd_1} B$, we would find that $(cd_1 \oplus cd_2) \oplus cd_3$ is undefined, but at the same time $cd_1 \oplus (cd_2 \oplus cd_3)$ might be a syntactically valid CD.

We have given ample evidence that it is not sufficient to ensure that a composite CD produced by the merge operation is a refinement (or expansion) of its component diagrams but that additional implementation-specific requirements must be taken into account. It is paramount for any implementation of a merge operator to detect and flag such conflicts.

6. Implementation and Evaluation

We have implemented the merge operator described in section 5.2 in the `CDMerge`-tool which is part of the larger `CD4Analysis`-project developed at the Chair of Software Engineering at RWTH Aachen University. The project is publicly available on GitHub[3]. `CDMerge` can be used as Java-library via the public methods that are provided by the `CDMerge` class or as a CLI-Tool via the `MCCD.jar`.

Consider the two component-CDs from our motivating example in section 3. In order to apply the merge, we first have to translate the input-CDs into the textual notation of `CD4Analysis` (Schindler 2012) and save each of them as a `cd`-file. We have already included the files `Teaching.cd` and `Management.cd` in the `doc` folder of the project. Their contents is listed in listing 1 and listing 2, respectively.

```
classdiagram Teaching {
  class Professor{
    String name;
    String mail;
  }
  class Lecture {
    int credits;
  }
  class Student {
    long matrNr;
  }
  association [1] Professor (lecturer)<->(holds) Lecture;
  association attendance [*] Lecture <- Student;
}
```

Listing 1 Teaching.cd. Textual representation of fig. 1

```
classdiagram Management {
  abstract class Employee {
    String name;
  }
  class Professor extends Employee{
    int salary;
  }
  class Lecture{
    int courseID;
  }
  association Professor (lecturer) -> (holds) Lecture [1..*]
    ;
}
```

Listing 2 Management.cd. Textual representation of fig. 2

By executing the `MCCD.jar` with the the following CLI-command, the two CDs`Teaching.cd` and `Management.cd` are merged. The resulting CD is then pretty-printed (`-pp`) to the console. The runtime of the merge process itself is negligible compared to the parsing of the input-CDs.

```
java -jar MCCD.jar -i Teaching.cd \
                --merge Management.cd -pp
```

More configuration options are documented in the README. A link for downloading the `MCCD.jar` is also provided. The result of the merge is listed in listing 3.

6.1. Case Study: MaCoCo

`CDMerge` was evaluated using the CDs of the *Ma*nagement *Co*ckpit for university chair's *Co*ntrolling (MaCoCo)[4] project.

[3] `https://github.com/MontiCore/cd4analysis`
[4] `https://www.se-rwth.de/projects/#MaCoCo`

```
classdiagram Merge {
  class Professor extends Employee {
    int salary;
    String mail;
  }
  class Lecture {
    int courseID;
    int credits;
  }
  abstract class Employee {
    String name;
  }
  class Student {
    long matrNr;
  }
  association  [1] Professor  (lecturer)<->(holds) Lecture
    [1..*];
  association attendance [*] Lecture  <- Student;
}
```

Listing 3 Merge result on the console

MaCoCo handles financial management specific to needs of university chairs, e.g. accounting of third-party funds. It is used by more than 180 chairs at the RWTH Aachen university with more than 1400 users in total.

The project uses a Model-Driven Engineering (MDE) approach to generate large parts of the application code. (Gerasimov et al. 2021; Adam et al. 2020; Gerasimov et al. 2020) describe the toolchain in more detail. The primary artifact is a single CD[5] with 1.141 text lines defining more than 100 classes and enums. The system consists of four subdomains: 1) Financial 2) Staff 3) Project and 4) Application Settings. Given the size of the model it is not easy to comprehend and alterations of one component require thorough manual checking to avoid inconsistencies.

A single holistic CD is needed for code generation. However, this CD need not be a source-model itself, but instead could be composed on demand. Therefore, we decided to decompose the monolithic CD into smaller CDs corresponding to the subdomains. The process was straightforward, since the monolithic CD already utilizes comments to segment classes and association into the aforementioned subdomains, each of them having tight internal but low external coupling. The subdomain-CDs are smaller, with about 250-300 lines of code. The merge process takes only a few seconds on standard office hardware.

We were able to proof that `CDMerge` produces a full CD that is semantically equivalent to the original model by using a CD differencing tool (Maoz et al. 2011; Nachmann et al. 2022). The evaluation shows that `CDMerge` is applicable to large CDs that are used in real-life MDE projects. Data models of subdomains can now be maintained individually while consistency is guaranteed by `CDMerge`.

6.2. Runtime Data Integration of Merged MDD Components

So far we have addressed the first research question by defining a sound merge operator in section 5 and evaluated it's applicability in the previous section. In order to address the second research question regarding the integration of preexisting software components, we have evaluated the applicability and limitations of `CDMerge` based component integration with a proof-of-concept. The objective was to integrate three software sub-components according to their respective data models. The integration of the data architectures by `CDMerge` ensured the interoperability of data structures at design time, but we identified constraints and issues that arise from integrating the corresponding run-time data which will be discussed in the following.

For integration purposes it is mandatory that each component's CD precisely specifies binding agreements for the data structure and the corresponding implementation of the respective component. Analogous to a component's API that serves as a functional contract, we could consider a CD as an ADI, an Application Data Interface specification. Any form of communication between software components relies on compatible data structures. Even if components do not actively communicate with each other, they usually operate on parts of a common data model. It is thus paramount that any implementation of the software component strictly obeys the data-type specifications of the CD. In particular, no alteration, extension or refinement of data types and their associations are allowed in both the generated and handwritten code. In that sense, the component's CD defines a *closed-world* data model for each sub-component, expect of underspecified model elements such as cardinality or name of associations. This restriction is necessary to allow `CDMerge` to check the consistency of the involved models. However, this does not impose a severe impediment on the component's design and implementation, it just requires thorough documentation or relying on code generation for the component's data model. `CDMerge` can then be included in, *e.g.,* continuous integration setups to provide feedback on any inconsistency or potential conflicts between data models of the software components. As `CDMerge` itself operates under the *open-world* assumption, a component's underspecification is reduced by introducing elements from other components or specifying association names, directions, role-names, and cardinalities based on matching associations. This will not cause issues as long as the component's implementation did not make any assumptions that are not reflected in the respective CD. When we consider the example of the underspecified cardinality of an association, an implementation can assume that objects can be retrieved or stored via corresponding links, but it needs to be robust in terms of storage buffer sizes as well as the handling of missing objects (`null` pointers).

Next, in order to specify time constraints, data integration during runtime must be considered. Take the class `Professor` from the running example: any object instance that refers to the same entity should be consistent across all software components. If an attribute value is changed in one software component, this change must be propagated to all other software components that deal with this instance.

One building block of this proof-of-concept is MontiDex, a MDD tool that generates data explorer views from CDs as Java applications (Roth 2017). As depicted in fig. 9 each component generated by MontiDex has a three-tier architecture that is comprised of a *GUI*, a *Business Logic* and a *Data Access Layer*. We used MontiDex to generate three applications that reflect parts

[5] Available online: `https://zenodo.org/record/6422355`

of a university management system. The two components to integrate were more comprehensive versions of the *Teaching* and *Management* components presented in our motivating example in section 3 which were integrated into a *University* application according to the approach sketched in fig. 3. The objectives were to (1) ensure that each component can function as a standalone application and (2) implement a particular view of the integrated software. For this purpose, we have implemented a middleware known as the *Entity Management Framework* which integrates with the MontiDex code generator and generates glue code that manages the consistency of runtime objects across MDD components.

We connected the three components to a central data storage with a schema matching the merged data model generated by `CDMerge` that is managed by the implemented middleware. Each application can manipulate data instances according to the local data model and the component integration is done via the data access layer. All CRUD[6] operations performed by any of the components are then delegated to and handled by the middleware.

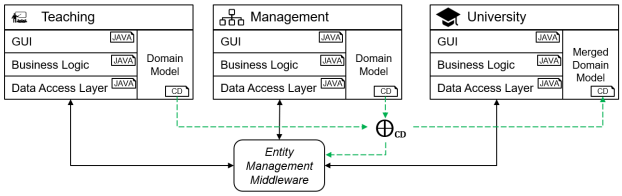

Figure 9 Software Components *Teaching* and *Management*, the integrated component *University* and the runtime entity management middleware.

Middleware and components must be able to deal with situations where attributes or object links are not yet fully initialized or object structures become compromised (*e.g.,* when objects are deleted elsewhere). As shown in fig. 10, the object structure modelled in `OD University` becomes invalid as soon as the *Teaching* module deletes the `Lecture`-object. This is permissible for the `Teaching`-component, as the cardinality is underspecified, but not for the *Management*- and *University*-components. `CDMerge` produces a warning in such cases and the middleware prevents the deletion of mandatory instances.

We identified four basic constraints for runtime compatibility of an integrated data model:

1. *Integrity*: Contradictory declarations in the CDs of the individual modules must be uncovered and fixed before runtime integration.

2. *Conformance*: A data instance must conform to all component data models w.r.t. open-world semantics, *i.e.,* attributes have correct data types and specifications of the associations, like cardinality constraints, are not violated. Any manipulation of a data instance that leads to an invalid state of the entire system must be prevented.

[6] Create, Read, Update, Delete

3. *Unique referenceability*: Data instances must be accessible and uniquely referenceable in all modules that use this data type. Changes to a data instance are visible to other modules as long as the corresponding attributes and associations are known locally.

4. *Consistency*: If a data instance is manipulated, the change and the new object state are propagated to all components that refer to this data instance.

The first two constraints are ensured by the formal soundness of the merge operator as well as by detecting the merge conflicts outlined in section 5.3. However, for runtime integration we have to consider multiple implementations and hence runtime polymorphism of a class, one for the data model of each component. We identified two ways to handle this situation:

1. Code transformation that replaces each data type declarations in every software component with a data type declaration of the merged model.

2. Runtime transformation of one datatype representation to another.

The first option is unfavourable as it would require recompilation of each software component every time a data model of any component is changed. It would also impose the inflated data structures of the merged data model on all software components, as well as risk propagation of potentially sensitive information. That leaves us with the option of transforming object structures at run-time. Since all variants of every data type are known at design time, the type transformers and necessary adapters/wrappers can be derived and generated from the component-CDs.

Ensuring the third constraint, *unique referenceability*, is challenging. The system needs to be able to trace data objects that refer to the same entity. This requires primary identifiers for object instances. One option, derived from the application domain, are identifiers created from a common set of attribute values for a datatype that are known in all components or unique data links to other referable instances. Consider the motivating example again. For an instance of `Professor` the identifier could be the attribute `String name` (see figs. 1 and 2). For `Lecture` there is no common attribute across the components. Here, the merge tool can output a warning, that object instances of a `Lecture` might not be unambiguously referable and duplicate data instances have to be resolved manually. Alternatively, it must be guaranteed that instances of this class can be created only in the integrated component and the sub-components can only view or edit data.

Assuming that either of the identifiers is used or that objects are only created and stored at a single location, changes can be propagated automatically to the central data storage and across all components using well known *Observer* and *Mediator* patterns (Gamma et al. 1995) that are generated and attached to each entity derived from the input CDs. This ensures the fourth constraint, *consistency*.

Lastly, we have to deal with incomplete data initialisation. This situation is depicted in fig. 10 which shows three UML/P Object Diagrams (ODs) that represent instances of their

respective CD. We can see that an object that corresponds to the merged data model *University* can easily be transformed into objects of the respective component-CDs by omitting unspecified elements, *e.g.,* the `p3:Professor` can be transformed into `p2:Professor` by removing the attribute `salary`. This is always possible as the composite-CD is always an expansion of its components. On the other hand, `p1:Professor` cannot be easily transformed to a valid object for the *Management*-CD as the the attribute `salary` as well as the required link to a `Lecture`-object are missing (note the cardinality `1..*` in fig. 2). One possible solution is to use only the composite-CD as a template for object creation. Whenever a data object needs to be created, the creation is delegated to a factory in a middleware and any mandatory but unspecified attributes or object links are created with default values. The middleware then needs to keep track of partially initialized objects and their default values.

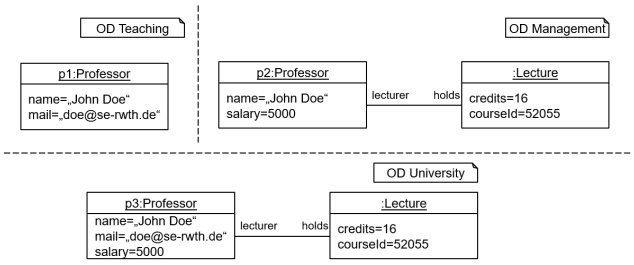

Figure 10 Object Diagrams illustrating instances of three possible type variants of the class `Professor` according to their respective component-CDs.

7. Conclusion and Outlook

We have defined and implemented a sound merge operator that allows merging underspecified CDs based on a formal open-world semantics. Furthermore, we have discussed potential merge conflicts and proposed variants of the operator to address implementation-oriented aspects of software composition that are not covered by the formal notion of semantic refinement.

To answer our first research question, we believe that an expansion-based notion of soundness (cf. definition 5.1) is an appropriate choice for the formal definition of a merge operator, as it ensures refinement in the open-world, permits reducing under-specification, and prevents altering specified cardinalities. The final property mentioned preserves "write"-promises that are not considered in the formal semantics of CDs, but that are relevant for manual extension of functionality.

As for the second research question, our approach can detect conflicts at the data level, as well as basic problems at the operational level early on and is well-suited for continuous integration. However, the formal notion of a semantically sound merge does not guarantee re-usability of preexisting software components in all cases. Similarly, the merge conflicts outlined in section 5.3 are necessary but not sufficient to cover implementation-specific aspects. A dedicated generator-infrastructure and additional considerations for the target language are needed. Moreover, we allow variations of the merge operator that may violate certain aspects of formal soundness in order to accommodate implementation-specific needs.

We have evaluated our approach in a case study where we merged CDs of system components into a holistic CD of the whole system. Here, `CDMerge` proved to be able to facilitate valid composition of MDD system architectures. In addition, we have examined runtime implications in a feasibility study using a prototype implementation of an *Entity Management Framework* that allows for late binding of software components and integration of existing software modules as long as the integrated data model is consistent. We have identified challenges regarding referenceability that occur for domain instances in run-time data integration, and intend to address them in a future publication. In a next step, we aim to generalize this approach using a code generator that takes the component-CDs as inputs and generates the required glue code to integrate the software modules. Our hope is that this facilitates reuse of ready implemented, model-based components as building blocks for MDD.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 250902306

References

Adam, K., Michael, J., Netz, L., Rumpe, B., & Varga, S. (2020, May). Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 years emisa: Digital ecosystems of the future: Methodology, techniques and applications (emisa'19)* (Vol. P-304, p. 59-66). Gesellschaft für Informatik e.V.

Aßmann, U. (2003). Invasive software composition. In *Invasive software composition* (pp. 107–145). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-662-05082-8_4

Batini, C., Lenzerini, M., & Navathe, S. B. (1986, dec). A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, *18*(4), 323–364. doi: 10.1145/27633.27634

Boronat, A., Carsí, J. A., Ramos, I., & Letelier, P. (2007). Formal model merging applied to class diagram integration. *Electronic Notes in Theoretical Computer Science*, *166*, 5-26. (Proceedings of the ERCIM Working Group on Software Evolution (2006))

Dalibor, M., Michael, J., Rumpe, B., Varga, S., & Wortmann, A. (2020, October). Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, & H. C. Mayr (Eds.), *Conceptual modeling* (p. 377-387). Springer International Publishing.

Dingel, J., Diskin, Z., & Zito, A. (2008). Understanding and improving uml package merge. *Software & Systems Modeling*, *7*(4), 443–467. doi: \10.1007/s10270-007-0073-9

Fahrenberg, U., Acher, M., Legay, A., & Wąsowski, A. (2014). Sound merging and differencing for class diagrams. In S. Gnesi & A. Rensink (Eds.), *Fundamental approaches to software engineering* (pp. 63–78). Berlin, Heidelberg: Springer Berlin Heidelberg.

France, R., & Rumpe, B. (2007, May). Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, 37-54.

Gamma, E., Helm, R., Johnson, R., Johnson, R. E., Vlissides, J., et al. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

Gerasimov, A., Michael, J., Netz, L., & Rumpe, B. (2021, March). Agile Generator-Based GUI Modeling for Information Systems. In A. Dahanayake, O. Pastor, & B. Thalheim (Eds.), *Modelling to program (m2p)* (p. 113-126). Springer.

Gerasimov, A., Michael, J., Netz, L., Rumpe, B., & Varga, S. (2020, August). Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In B. Anderson, J. Thatcher, & R. Meservy (Eds.), *25th americas conference on information systems (amcis 2020)* (p. 1-10). Association for Information Systems (AIS).

Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Mir Seyed Nazari, P., Müller, K., ... Wortmann, A. (2015). Integration of Handwritten and Generated Object-Oriented Code. In *Model-driven engineering and software development* (Vol. 580, p. 112-132). Springer.

Harel, D., & Rumpe, B. (2004, October). Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, *37*(10), 64-72. Retrieved from `http://www.se-rwth.de/staff/rumpe/publications20042008/Meaningful-Modeling-Whats-the-Semantics-of-Semantics.pdf`

Johannes, J., & Aßmann, U. (2010). Concern-based (de)composition of model-driven software development processes. In D. C. Petriu, N. Rouquette, & Ø. Haugen (Eds.), *Model driven engineering languages and systems* (pp. 47–62). Berlin, Heidelberg: Springer Berlin Heidelberg.

Kolovos, D. S., Paige, R. F., & Polack, F. A. C. (2006). Merging models with the epsilon merging language (eml). In O. Nierstrasz, J. Whittle, D. Harel, & G. Reggio (Eds.), *Model driven engineering languages and systems* (pp. 215–229). Berlin, Heidelberg: Springer Berlin Heidelberg.

Lutz, R., Wurfel, D., & Diehl, S. (2011). How humans merge uml-models. In *2011 international symposium on empirical software engineering and measurement* (p. 177-186). doi: 10.1109/ESEM.2011.26

Maoz, S., Ringert, J. O., & Rumpe, B. (2011). CDDiff: Semantic Differencing for Class Diagrams. In M. Mezini (Ed.), *Ecoop 2011 - object-oriented programming* (p. 230-254). Springer Berlin Heidelberg.

Meier, J., Werner, C., Klare, H., Tunjic, C., Aßmann, U., Atkinson, C., ... Winter, A. (2020). Classifying approaches for constructing single underlying models. In S. Hammoudi, L. F. Pires, & B. Selić (Eds.), *Model-driven engineering and software development* (Vol. 1161, pp. 350–375). Cham: Springer International Publishing. doi: \10.1007/978-3-030-37873-8_15

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, *28*(5), 449-462. doi: 10.1109/TSE.2002.1000449

Michael, J., Nachmann, I., Netz, L., Rumpe, B., & Stüber, S. (2022, June). Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines. In *Modellierung 2022* (p. 33-48). Gesellschaft für Informatik.

Nachmann, I., Rumpe, B., Stachon, M., & Sebastian, S. (2022, June). Open-World Loose Semantics of Class Diagrams as Basis for Semantic Differences. In *Modellierung 2022* (p. 111-127). Gesellschaft für Informatik.

Nagl, M. (1990). Softwaretechnik - methodisches programmieren im großen. In *Springer compass.*

Reiß, D. (2016). *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Shaker Verlag.

Roth, A. (2017). *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Shaker Verlag.

Rumpe, B. (2011). *Modellierung mit UML, 2te Auflage*. Springer Berlin.

Schindler, M. (2012). *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Shaker Verlag.

Wile, D. (2003). Lessons learned from real dsl experiments. In *36th annual hawaii international conference on system sciences, 2003. proceedings of the* (p. 10 pp.-). doi: 10.1109/HICSS.2003.1174893

Wimmer, M., Schauerhuber, A., Kappel, G., Retschitzegger, W., Schwinger, W., & Kapsammer, E. (2011, oct). A survey on uml-based aspect-oriented design modeling. , *43*(4).

About the authors

Achim Lindt is a research assistant and Ph.D. candidate at the Chair of Software Engineering at RWTH Aachen University. His research topic is model based integration of software components, model reuse and model repositories. You can contact the author at lindt@se-rwth.de or visit www.se-rwth.de.

Bernhard Rumpe is Professor at RWTH Aachen University, Germany and head of the Chair of Software Engineering. His main interests are rigorous and practical software and system development methods based on adequate modeling techniques. This includes agile development methods as well as model engineering based on UML/SysML-like notations and domain-specific languages. You can contact the author at rumpe@se-rwth.de or visit www.se-rwth.de.

Max Stachon is a research assistant and Ph.D. candidate at the Chair of Software Engineering at RWTH Aachen University. His research focuses on model semantics, refinement and difference analysis. You can contact the author at stachon@se-rwth.de or visit www.se-rwth.de.

Sebastian Stüber is a research assistant and Ph.D. candidate at the Chair of Software Engineering at RWTH Aachen University. His research focuses on model semantics, refinement, difference analysis and formal verification. You can contact the author at stueber(at)se-rwth.de or visit www.se-rwth.de.