On Proving the Correctness of Refactoring Class Diagrams of MDE Metamodels

NAJD ALTOYAN and DON BATORY, The University of Texas at Austin, USA

Model Driven Engineering (MDE) is a general-purpose engineering methodology to elevate system design, maintenance, and analysis to corresponding activities on models. Models (graphical and/or textual) of a target application are automatically transformed into source code, performance models, Promela files (for model checking), and so on for system analysis and construction.

Models are instances of *metamodels*. One form an MDE metamodel can take is a [class diagram, constraints] pair: the class diagram defines all object diagrams that could be metamodel instances; object constraint language (OCL) constraints eliminate semantically undesirable instances.

A metamodel *refactoring* is an invertible semantics-preserving co-transformation, i.e., it transforms both a metamodel *and* its models without losing data. This article addresses a subproblem of metamodel refactoring: how to prove the correctness of refactorings of class diagrams without OCL constraints using the Coq Proof Assistant.

CCS Concepts: • **Software and its engineering → Functionality**; • **Theory of computation → Program semantics;**

Additional Key Words and Phrases: Class diagram refactorings, object diagram refactorings, Coq

ACM Reference format:
Najd Altoyan and Don Batory. 2023. On Proving the Correctness of Refactoring Class Diagrams of MDE Metamodels. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 44 (March 2023), 42 pages.
https://doi.org/10.1145/3549541

1 INTRODUCTION

Model Driven Engineering (**MDE**) is a general-purpose engineering methodology for system analysis, reasoning, change management, and other activities [23]. An MDE *model* (possibly plural) is a specification of a target application. A model can be transformed into a performance model, Promela file (for model checking), source code, and so on for system analysis and construction. Models are instances of a *metamodel*, sometimes called a ***Domain Specific Language*** (**DSL**) [99]. Models and metamodels can be graphical (class diagrams, state charts), textual (sentences of a grammar, code fragments, object constraint language (OCL) constraints), or an integration of both [5, 22, 23].

Najd Altoyan also with King Abdulaziz City for Science and Technology (KACST).
Altoyan was supported by King Abdulaziz City for Science and Technology (KACST). This work was also supported by NSF Grant no. 26-1005-25 (Award no. 1212683).
Authors' address: N. Altoyan and D. Batory, Department of Computer Science, University of Texas at Austin, Austin, Texas; emails: naltoyan@utexas.edu, batory@cs.utexas.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
1049-331X/2023/03-ART44 $15.00
https://doi.org/10.1145/3549541

Like programs, metamodels gradually evolve for reasons of maintenance, simplification, and accommodation of new functionalities. Also like programs [14, 57, 78], refactorings are suited for these tasks. A *refactoring* is a semantics-preserving transformation. Today's mainstream (Java) integrated development environments (IDEs) help their users by offering a wealth of refactorings [31, 38, 43, 68, 75]. There is ample evidence that MDE architects want a comparable level of support for metamodel refactorings on MDE platforms [33, 52, 63, 71].

A common representation of an MDE metamodel $\mathbb{m}$ is a [class diagram, constraints] pair: a ***UML class diagram*** (**umlCD**) *cd* defines all ***Object Diagrams*** (**ODs**) that could be metamodel instances; **OCL** (***Object Constraint Language***) constraints k eliminate semantically undesirable instances. We write $\mathbb{m}$=$[cd, k]$. Let $\boldsymbol{R}$ be a metamodel refactoring. $\boldsymbol{R}$ transforms metamodel $\mathbb{m}$ into an equivalent metamodel $\mathbb{m}'$=$[cd', k']$. A distributivity law—a refactoring distributes over a metamodel's components—relates $\mathbb{m}$ and $\mathbb{m}'$:

$$\boldsymbol{R}(\mathbb{m}) \;=\; \boldsymbol{R}([cd, k]) \;=\; [\boldsymbol{R}(cd), \boldsymbol{R}(k)] \;=\; [cd', k'] \;=\; \mathbb{m}'. \tag{1}$$

What seems not to be well-known is that the inverse of a refactoring is also a refactoring. Thus,

$$\boldsymbol{R}^{-1}(\mathbb{m}') \;=\; \boldsymbol{R}^{-1}([cd', k']) \;=\; [\boldsymbol{R}^{-1}(cd'), \boldsymbol{R}^{-1}(k')] \;=\; [cd, k] \;=\; \mathbb{m}. \tag{2}$$

That is, $\mathbb{m}$ and $\mathbb{m}'$ are *equivalent* w.r.t. $\boldsymbol{R}$. Observe that $\boldsymbol{R}(\mathbb{m})$ is a coordinated pair of refactorings: a umlCD refactoring $\boldsymbol{R}(\mathtt{cd})$ *and* an OCL constraints refactoring $\boldsymbol{R}(\mathtt{k})$.

A common restriction on metamodels is that their umlCDs have no interfaces, statics, and methods. Such umlCDs define only data relationships, which enables them to be translated to database schemas and their ODs to databases [11, 13, 37]. We focus on these umlCDs and their ODs in this article.

Correctness is an important property of refactorings. The *Eclipse* ***Java Development Tool*** (**JDT**) is among the most advanced IDEs and offers frequently used refactorings. Yet it is known that JDT refactorings can alter program behavior or produce uncompilable code [46, 81]. Other major Java IDEs including NetBeans, Oracle JDeveloper, and IntelliJ IDEA are no different [47]. Lacker et al. [49] reported that there are 5,045 refactoring-related bug reports in the Eclipse bug report website and that 18.4% of the reported bugs will never be fixed.

Correctness of MDE refactorings are also important, but have the advantage that umlCD refactorings are simpler than Java refactorings. Still, there are difficulties. umlCDs semantics are not uniform across MDE platforms [69, 87], and so too are their encodings as relational databases [18, 19, 61, 65].

This article is on the correctness of umlCD refactorings. The semantics of the few umlCD features that we use are consistent with early UML standards [32, 83] and that of typical research papers in MDE. Also, our mappings of models to main-memory, text-file-persistent relational databases are direct. For these reasons, our approach and results should be transferable to other MDE platforms and UML tools.

1.1 Class Diagram Refactorings are Co-Transformations

A *co-transformation* is a transformation of a type and its instances [91]. umlCD refactorings are co-transformations. We are interested in the verification of ***minimal umlCD refactorings*** (***minRefs***) that use a small umlCD with only the essential elements to capture a refactoring's essence. A *minRef* is $\boldsymbol{R}_\ominus$:$\{cd\}$$\rightarrow$$\{cd'\}$, where $\ominus$ labels a *minRef*, *cd* is its minimal input umlCD, and *cd′* is its minimal output umlCD. $\boldsymbol{R}_\ominus$ must satisfy the round-tripping constraints of Equations (1) and (2): $\boldsymbol{R}_\ominus$ converts *cd* to *cd′* and $\boldsymbol{R}_\ominus^{-1}$ restores *cd* from *cd′*:

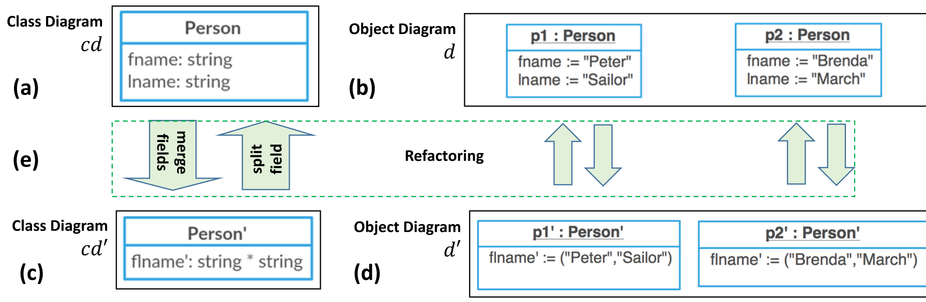


Fig. 1. Class diagrams, object diagrams, and their correspondences.

$$\left(cd = \boldsymbol{R}_{\ominus}^{-1} \cdot \boldsymbol{R}_{\ominus}(cd)\right) \quad \bigwedge \quad \left(cd' = \boldsymbol{R}_{\ominus} \cdot \boldsymbol{R}_{\ominus}^{-1}(cd')\right). \tag{3}$$

Further, $\boldsymbol{R}_{\ominus}$ also refactors models and preserves their semantics. That is, given any OD d of umlCD cd, round-tripping recovers d, and similarly for $\boldsymbol{R}_{\ominus}^{-1}$:

$$\left(\forall d \in cd: \; d = \boldsymbol{R}_{\ominus}^{-1} \cdot \boldsymbol{R}_{\ominus}(d)\right) \quad \bigwedge \quad \left(\forall d' \in cd': \; d' = \boldsymbol{R}_{\ominus} \cdot \boldsymbol{R}_{\ominus}^{-1}(d')\right). \tag{4}$$

We show how to prove Equations (3) and (4) using the Coq Proof Assistant in this article [21].

1.2 Examples of Class Diagram Refactoring

Figure 1(a) is a umlCD cd with one class, `Person`, having two `string` *attributes*: first name (`fname`) and last name (`lname`). Figure 1(b) is an OD d of cd with two `Person` instances, "`Peter Sailor`" and "`Brenda March`" [1]. Another umlCD, cd', is Figure 1(c). It differs from cd by the composite attribute (`flname'`) replacing `fname` and `lname`. Figure 1(d) shows OD d' of cd', also with two *Person'* instances.

As Figure 1(e) suggests, cd' and d' are refactorings of cd and d, and vice versa. The *minRefs* that accomplish this, $\boldsymbol{mergefields}_{\ominus}$:$\{cd\}\to\{cd'\}$ and $\boldsymbol{splitfield}_{\ominus}$:$\{cd'\}\to\{cd\}$, satisfy Equations (3) and (4):

$$cd = \boldsymbol{splitField}_{\ominus}(\boldsymbol{mergeFields}_{\ominus}(cd)) \quad \wedge \quad cd' = \boldsymbol{mergeFields}_{\ominus}(\boldsymbol{splitField}_{\ominus}(cd')), \tag{5}$$

$$\forall d \in cd \; : \; d = \boldsymbol{splitField}_{\ominus}(\boldsymbol{mergeFields}_{\ominus}(d)) \qquad \wedge$$

$$\forall d' \in cd' \; : \; d' = \boldsymbol{mergeFields}_{\ominus}(\boldsymbol{splitField}_{\ominus}(d')). \tag{6}$$

Equations (5) and (6) must be proven.

It is worth considering what is *not* a refactoring. PushDown field and its inverse PullUp field are usually *edits*, not refactorings. They *are* refactorings only when superclass A is abstract. Consider Figure 2. Class A is not abstract, meaning it can have instances that do not belong to any of A's subclasses. When field $A.f$ is pushed down, the fields of A subclasses are unchanged. However, A objects lose their f field *and* their f values. The PushDown field of this example loses data and therefore is not a refactoring. Neither is PullUp field in general, as it must add missing data to its subclass objects and it too is not a refactoring.

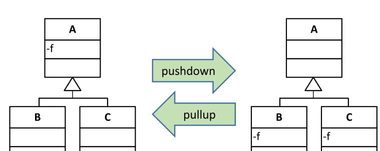


Fig. 2. PushDown and PullUp field.

1.3 On the Non-Uniqueness of *minRef*s

Not every refactoring has a unique *minRef*; there could be several. Figure 3 shows three *minRef*s for PushDown; all A classes are abstract. Figure 3(a) pushes down field $A.x$ into a single subclass B. Figure 3(b) pushes down multiple fields x, y into B. And Figure 3(c) pushes down field x into multiple subclasses. These variations can be combined. Any could be chosen, but we found choosing the least complicated (Figure 3(a)) makes *minRef* proofs easier.

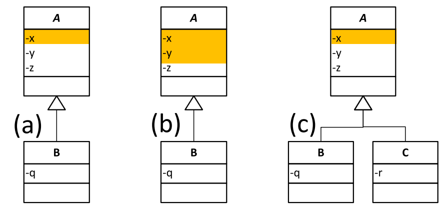


Fig. 3. *minRef*s for ***PushDown***$_\ominus$.

1.4 Big Picture of This Article

umlCD refactorings are considerably larger than *minRef*s in practice. How does our work on *minRef*s contribute to larger refactorings? We have four answers.

First, *minRef*s are a good starting point. Two extensions (generalizations) of *minRef*s scale refactorings to that expected by umlCD architects. These extensions are explained in Section 5. Call these extensions $\mathcal{X}$ and $\mathcal{Y}$, as their details are irrelevant now. (Example: $\mathcal{X}$ could be PushDown multiple fields in Figure 3(b) and $\mathcal{Y}$ could be A with multiple subclasses). It will be evident from this article that verifying *minRef*s is sufficiently complicated in Coq. Our experience has convinced us that tackling *all* challenges at once—verifying *minRef*s with extensions $\mathcal{X}$ *and* $\mathcal{Y}$—would be lethal (too daunting to achieve). Instead, stepwise extensions of *minRef*s is a practical way to scale correctness proofs [8, 9, 12, 28, 86]. Meaning: verify a *minRef*, then generalize the proof to support $\mathcal{X}$, and then do the same for $\mathcal{Y}$. More on this in Section 5.

Second, contemporary Java IDEs offer a wealth of primitive refactorings for programmers to use. It is not well-known that most (not all) design patterns, as in the Gang-of-Four Text [34], are composite refactorings [8, 45, 46, 92]. That is, by scripting a series of primitive refactorings, a program without a design pattern (e.g., visitor) can be automatically refactored into one with that pattern. This is a practical form of scaling primitive refactorings, but not yet their verification.

Third, composing refactorings in Category Theory (which we discuss shortly) is simple: it is function composition, as refactorings are functions. In practice, such functions become a refactoring *script* (read: Java method) that uses local variables and invokes primitive refactorings (which may themselves be scripts) directly, conditionally, or in loops (where the same refactorings are invoked with different arguments on each loop iteration) [45, 46]. The theorems to prove are the same, Equations (3) and (4), except each $\boldsymbol{R}_\ominus$ is now a script. We believe Coq scales to this task, but admit in the Conclusions that Coq was *not* the ideal prover for us to use in this article and in future work on *minRef* generalization.

Fourth, prior work on database metamodel management [18, 19, 59, 61] suggests a rather different and potentially easier way to address umlCD refactoring correctness. We explain the idea in Section 5.3.

1.5 Article Organization

Every umlCD refactoring has a minimal definition (*minRef*): a simple and paired-down case to study. Some *minRef*s have no constraints; most have cardinality and/or uniqueness constraints (see Section 2.4). We call each such constraint a *minimal constraint* (*minCon*). *minRef*s with *minCon*s are more difficult to prove correct than those without.

*minCon*s differ from OCL constraints. *minCon*s are essential for verifying the correctness of a *minRef* and are preconditions to apply a *minRef* to a umlCD. OCL constraints serve a different purpose: they

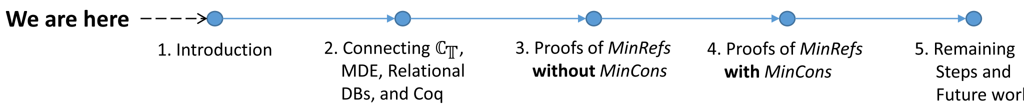


Fig. 4. Roadmap to sections of this article.

eliminate semantically unwanted ODs permitted by a umlCD. Each requires different techniques for verification.

1.6 Approach and Contributions

Approach. *Category Theory* ($\mathbb{C}_{\mathbb{T}}$) provides a formal and visual foundation that is central to our work. Relational databases are also essential:

(1) Metamodels with umlCDs (sans OCL constraints) are relational schemas, and their model instances are relational databases [13].
(2) Metamodel refactorings correspond to schema refactorings and model refactorings correspond to database refactorings.
(3) Database concepts, not MDE concepts, are close to the abstractions offered by the Coq Proof Assistant [21], the prover we use to verify *minRef*s in Sections 3 and 4.

Figure 4 is the roadmap to this article. Each node (section) progressively builds upon the results of prior sections. Tackling all challenges at once, we found, was unintelligible.

Contributions. The contributions of our article are as follows:

(1) umlCD refactorings define umlCD equivalences.
(2) Correctness proofs of *minRef*s, with and without *minCon*s, using Coq.
(3) How proofs of *minRef*s can be extended to larger refactorings.
(4) An outline of a future theorem prover for this line of work.
(5) Eight distinct *minRef*s that we have verified (Appendix G and [2]).
(6) A replication package with all Coq artifacts in this article is [3].

2 RELATING CATEGORIES, MDE, RELATIONAL DATABASES, AND COQ

2.1 Categories and MDE

Category Theory ($\mathbb{C}_{\mathbb{T}}$) is a theory of total functions, called *arrows*, that relate structures.

Structures. A *structure* is a data type that defines the data contained in its instances, but *without* operations. Every object in Java belongs to a structure called a class, and each class defines the attributes (data) that its objects maintain. (Yes, methods on objects are defined too, but structures are defined *without* methods/operations, much like C structs [44]). The umlCDs of this article are similar: umlCDs have no methods, statics, and interfaces; think of umlCDs as graphical database schemas [30, 84].

A structure may have a domain of instances. The domain of the Java *Integer* class is the set of all *Integer* objects. For schema $\mathbb{D}$, the domain of $\mathbb{D}$ is the set of all database instances of $\mathbb{D}$.

The domain of structure $\mathbb{S}$ is the set of all $\mathbb{S}$ instances and is depicted by a *cone-of-instances diagram*, Figure 5(a). $\mathbb{S}$ is the cone's apex and its domain is the base. Figure 5(a) shows three instances of $\mathbb{S}$ written as $\{s_1, s_2, s_3\}\subset\mathbb{S}$.

$\mathbb{S}$ can be an instance of a more abstract structure $\mathbb{T}$, recursing upwards to infinity, Figure 5(b). Practicality limits recursion to three

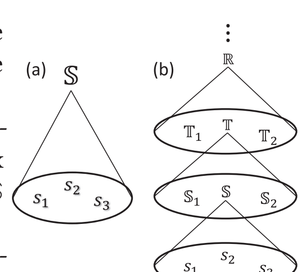


Fig. 5. Cone of instances.

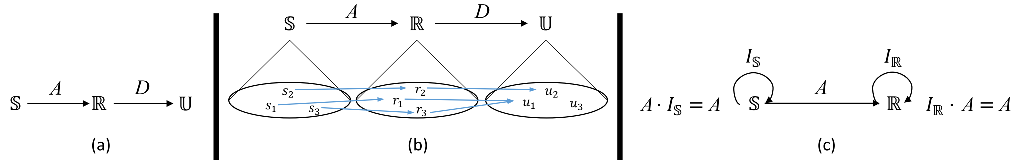


Fig. 6. External and internal diagrams, identity arrows.

levels in MDE, called the ***Meta-Object Facility*** (**MOF**) [8, 67]: *models* are instances of *metamodels*, and metamodels are instances of a single *meta-metamodel*.

Arrows. An *arrow* relates structures via a total function [95]. Arrow $A{:}\mathbb{S}{\to}\mathbb{R}$ in Figure 6(a) maps each $s{\in}\mathbb{S}$ to some $r{\in}\mathbb{R}$ and is drawn from A's domain $\mathbb{S}$ to A's co-domain $\mathbb{R}$. Arrows in MDE are called *transformations*.

A *directed multi-graph* allows multiple edges between nodes. An *external diagram* is a directed multi-graph where nodes are structures and arrows are directed edges. Figure 6(a) is an external diagram with three domains $\mathbb{S}$, $\mathbb{R}$, $\mathbb{U}$ and two arrows $A{:}\mathbb{S}{\to}\mathbb{R}$ and $D{:}\mathbb{R}{\to}\mathbb{U}$. An *internal diagram* is an external diagram with (a) cones of instances and (b) pairings of domain instances with co-domain instances that are consistent with the arrows of the external diagram. Figure 6(b) is an internal diagram where arrow A maps s_1 to r_1 and arrow D maps r_1 to u_1.

Arrow composition obeys three axioms; the first two are axioms of function composition:

(1) **Arrows compose.** If $A{:}\mathbb{S}{\to}\mathbb{R}$ and $D{:}\mathbb{R}{\to}\mathbb{U}$, then arrow $(D \cdot A){:}\mathbb{S}{\to}\mathbb{U}$ exists.
(2) **Arrows compose associatively.** $(E \cdot D) \cdot A = E \cdot (D \cdot A)$.
(3) **Identity Arrows.** Every structure $\mathbb{S}$ has an *identity arrow*: $I_{\mathbb{S}}{:}\mathbb{S}{\to}\mathbb{S}$, where $\forall s{\in}\mathbb{S} : I_{\mathbb{S}}(\mathtt{s}){=}\mathtt{s}$. Further, let $A{:}\mathbb{S}{\to}\mathbb{R}$. Then, $I_{\mathbb{R}} \cdot A = A$ and $A \cdot I_{\mathbb{S}} = A$ as in Figure 6(c).

Structure Equivalence. Structures $\mathbb{R}$ and $\mathbb{S}$ are *equivalent* or *isomorphic* if there are two arrows $\mathrm{T}{:}\mathbb{R}{\to}\mathbb{S}$ and $T^{-1}{:}\mathbb{S}{\to}\mathbb{R}$ such that T and T^{-1} are inverses of each other: $T \cdot T^{-1} = I_{\mathbb{S}}$ and $T^{-1} \cdot T = I_{\mathbb{R}}$.

Functors. The most sophisticated ideas on structures and arrows that we use are *functors*: arrows between external diagrams. Let $\mathcal{C}$ and $\mathcal{D}$ be external diagrams. Functor $\boldsymbol{F}:\mathcal{C}{\to}\mathcal{D}$ [70]:

— sends each structure $\mathbb{X}{\in}\mathcal{C}$ to structure $\boldsymbol{F}(\mathbb{X}){\in}\mathcal{D}$,
— sends each arrow $(A{:}\mathbb{X}{\to}\mathbb{Y}){\in}\mathcal{C}$ to arrow $(\boldsymbol{F}(A){:}\boldsymbol{F}(\mathbb{X}){\to}\boldsymbol{F}(\mathbb{Y})){\in}\mathcal{D}$,
— such that every arrow (given or composed) in $\mathcal{C}$ is preserved in $\mathcal{D}$.

Equivalent meanings of `X` "sends to" `Y` are as follows:

— `X` "is mapped to" `Y`, and
— `X` "is transformed to" `Y`.

Functor Example. Figure 7 shows external diagram $\mathcal{B}$ with two structures $\mathbb{X}$, $\mathbb{Y}$ and arrow $E{:}\mathbb{X}{\to}\mathbb{Y}$; identity arrows are implicit. External diagram $\mathcal{C}$ has three structures $\mathbb{S}$, $\mathbb{R}$, $\mathbb{U}$ and two arrows $K{:}\mathbb{S}{\to}\mathbb{R}$ and $L{:}\mathbb{R}{\to}\mathbb{U}$. Functor $\boldsymbol{H}:\mathcal{B}{\to}\mathcal{C}$ sends structures $\mathbb{X}$ to $\mathbb{S}$, $\mathbb{Y}$ to $\mathbb{R}$, and arrow E to K.

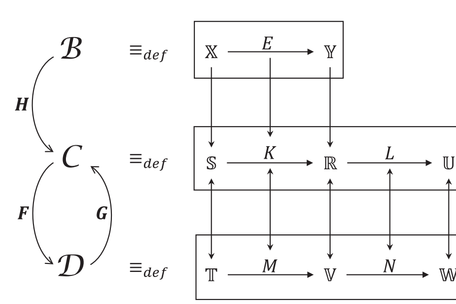


Fig. 7. Embedding and equivalence.

Diagram Equivalence. Let the *identity functor* for external diagram $\mathcal{C}$ be $\boldsymbol{I}_{\mathcal{C}}{:}\mathcal{C}{\to}\mathcal{C}$. Functor $\boldsymbol{F}:\mathcal{C}{\to}\mathcal{D}$ *embeds* external diagram $\mathcal{C}$ into $\mathcal{D}$, written $\mathcal{C}{\hookrightarrow}\mathcal{D}$. Further, $\mathcal{C}$ and $\mathcal{D}$ are *equivalent* or *isomorphic* if there exists two functors $\boldsymbol{F}{:}\mathcal{C}{\to}\mathcal{D}$ and $\boldsymbol{G}{:}\mathcal{D}{\to}\mathcal{C}$ such that $\boldsymbol{G}{\cdot}\boldsymbol{F} = \boldsymbol{I}_{\mathcal{C}}$ and $\boldsymbol{F}{\cdot}\boldsymbol{G} = \boldsymbol{I}_{\mathcal{D}}$. In other words, $\boldsymbol{F}$ and $\boldsymbol{G}$ are inverses of each other and their external diagrams embed each other, $\mathcal{C}{\hookrightarrow}\mathcal{D}$ and $\mathcal{D}{\hookrightarrow}\mathcal{C}$.

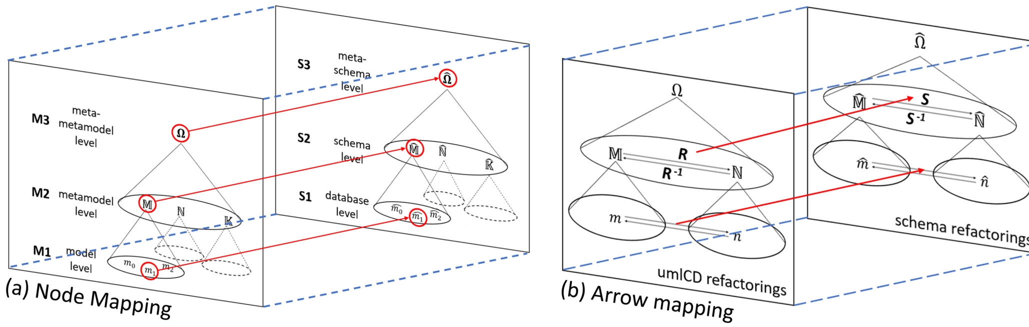


Fig. 9. MDE and database MOFs and their refactoring correspondences.

Equivalence Example. Figure 7 shows external diagram $\mathcal{D}$. Functor $\boldsymbol{F}: C \rightarrow \mathcal{D}$ sends domains $\mathbb{S}$ to $\mathbb{T}$, $\mathbb{R}$ to $\mathbb{V}$, $\mathbb{U}$ to $\mathbb{W}$, and arrows K to M and L to N. Functor $\boldsymbol{G}: \mathcal{D} \rightarrow C$ is the inverse of $\boldsymbol{F}$. Thus, external diagrams C and $\mathcal{D}$ are equivalent or isomorphic.

Category Theory. The above paragraphs are the core ideas of $\mathbb{C}_{\mathbb{T}}$ [70]. A *category* is another name for an external diagram; it is a set of structures and arrows as stated above. But in $\mathbb{C}_{\mathbb{T}}$ the term "object" is used instead of "structure." We use "structure" instead of "object" for the obvious reasons. MDE uses the terms "metamodel" and "class" for "structure" and "transformation" for "arrow."

We use $\mathbb{C}_{\mathbb{T}}$ as a language to explain umlCD refactorings. ***We use no deep theorems of $\mathbb{C}_{\mathbb{T}}$; only the terms, ideas, and axioms presented in this section and nothing more.***

Law Example. The "distributivity law" of Equations (1) and (2) can now be explained. See Appendix A.

2.2 MDE and Relational Databases

Figure 8 shows MOF has a single meta-metamodel Ω; metamodels are instances of Ω, and models are instances of metamodels.

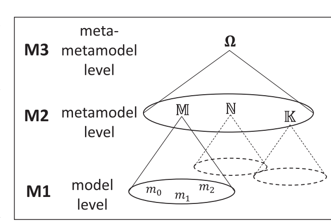


Fig. 8. The MOF hierarchy.

As said earlier, a umlCD of a metamodel is a graphical depiction of a relational database schema. Each class T of a umlCD has a corresponding relational table T: if $a_1 \ldots a_j$ are the attributes of class T, they are also columns of table T. Objects of class T are the tuples of table T. Every relational table has an explicit identifier column whose value is user- or tool-assigned [13, 37, 62], which corresponds to an object identifier in an MDE model. Just as there is class inheritance in umlCDs, there are corresponding inheritance relationships among tuple types and corresponding inheritance relationships among their tables [10]. Example: in a umlCD, `Mustang` is a subclass of `Horse` means `Mustang` is a sub-tuple-type of `Horse` and the table of `Mustang`s is a subtable of `Horse`s.

Database systems have their own MOF: there is a single metaschema $\widehat{\Omega}$, schemas are instances of $\widehat{\Omega}$, and all databases are instances of schemas. The functor of Figure 9(a) sends meta-metamodel Ω to metaschema $\widehat{\Omega}$, metamodel $\mathbb{M}$ to schema $\widehat{\mathbb{M}}$, and model (object-diagram) m to database $\widehat{m}$. Transforming a umlCD and OD into a schema and database with inheritance is well-known [11, 37, 64].

The functor of Figure 9(b) sends a umlCD refactoring $\boldsymbol{R}$ to a schema refactoring $\boldsymbol{S}$. For example, $\boldsymbol{S}$ splits a *Dog* table into a shorter *Dog* table connected to an *Owner* table, Figures 10(a)→(b). (In

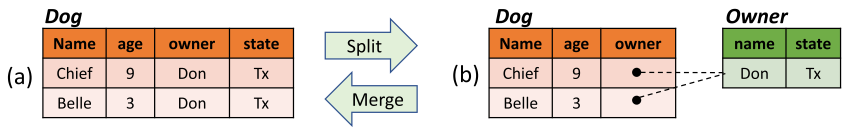


Fig. 10. Split and merge table refactorings.

database parlance, *Dog* is *normalized* [84]). $\boldsymbol{S}^{-1}$ restores the original *Dog* table, Figures 10(b)⟶(a). ***Refactoring a schema produces a new schema and a corresponding restructuring of its databases.*** Thus, schema refactoring $\boldsymbol{S}$ is a co-transformation.

Every umlCD *minRef* can be encoded as a schema *minRef*. $\mathbb{C}_{\mathbb{T}}$ tells us the theorems to prove. For each *minRef* $\boldsymbol{R}_{\ominus}$:$\{cd\}\longrightarrow\{cd'\}$, there is a corresponding database schema in *minRef* $\boldsymbol{S}_{\ominus}$:$\{s\}\longrightarrow\{s'\}$, where the following round-tripping theorems for schemas must be proven:

$$s \;=\; \boldsymbol{S}_{\ominus}^{-1}\cdot\, \boldsymbol{S}_{\ominus}(s) \qquad \bigwedge \qquad s' \;=\; \boldsymbol{S}_{\ominus}\cdot \boldsymbol{S}_{\ominus}^{-1}(s'). \tag{7}$$

And so too the round-tripping theorems of their databases:

$$\forall d\in s \;:\; d \;=\; \boldsymbol{S}_{\ominus}^{-1}\cdot\, \boldsymbol{S}_{\ominus}(d) \qquad \bigwedge \qquad \forall d'\in s' \;:\; d' \;=\; \boldsymbol{S}_{\ominus}\cdot \boldsymbol{S}_{\ominus}^{-1}(d'). \tag{8}$$

Appendix B explains implicit constraints of refactorings that we and others avoid.[1]

2.3 Overview of the Coq Proof Assistant

Coq is an interactive theorem prover based on a functional programming paradigm. The user guides the system until a proof is discharged [21].

Types. Coq defines two kinds of types: `Set` and `Prop`. As the name suggests, *Prop* is any propositional expression. Any type that is not a proposition falls under *Set*. *Set* includes types like strings (*string*), natural numbers (*nat*), and so on.

New types in Coq are encoded as *records*. A record is analogous to a umlCD class or a table definition in databases. Each record has a single constructor and a set of typed fields. A field's type can be a built-in type, user-defined type, function, or proposition. Consider the following Coq definition of class *Person*, Figure 11:

```
1  Record Person := mkPerson {    (* Person tuple constructor *)
2   fname : string;
3   lname : string;
4  }.
```

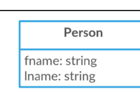


Fig. 11. *Person*.

Listing 1. Person class.

Here, *Person* is a new type and *mkPerson* is its constructor. Fields *fname* and *lname* represent a `Person`'s first and last names, respectively; both use the built-in type *string*. Fields are separated by semicolons (;) and statements are terminated with a period (.), Line 4.

Unlike ***Object-Oriented*** (**OO**) languages, Coq fields are functions. `fname` is a unary function that maps each `Person` to a `string` value, i.e., `fname:Person⟶string`. Thus to retrieve the `fname` for `Person` *p*, one writes (`fname` *p*), i.e., apply function `fname` to *p*, which is *p*.`fname()` in OO notation.

[1]Technically, since we use the Coq Proof Assistant to verify refactorings, we use yet another MOF translation from the database MOF of schemas and their databases to their corresponding MOF of Coq schemas and databases. We elide this extra layer of mapping. We explain our Coq encoding of a metaschema in Section 3.2, and our Coq encoding of a database refactoring in Appendix D.2.

Field values can be constrained. A *constraint* is a field whose type is `prop`. Constraint *notEmpty*, below, says the value of *lname* cannot be empty.

```
Record Person2 := mkPerson2 {  (* Person2 tuple constructor *)
 fname : string;
 lname : string;
 notEmpty : lname <> "";       (* <> means not equal *)
}.
```

When *Person*2 is instantiated, a proof must be supplied showing that the constraint holds. If no proof is given, a `Person2` *cannot* be instantiated. This is because Coq does not *evaluate* propositional expressions. More on this in Section 4.

Functions. Non-recursive functions are defined using the keyword *Definition*, followed by the name of the function, a list of parameters, and an optional return type. Parameters are usually surrounded by parentheses, followed by a colon (:) and the function's (single) return type. The body of a function, or the expression of a function, is given after the bind symbol (:=):

```
Definition toString (p: Person): string := (fname p) ++ " " ++ (lname p).
```

Listing 2. toString (pretty print) function.

Function *toString* takes a parameter *p* of type *Person* (Listing 1) and returns a `string` representation of *p*, called *pretty printing* a *Person* (Listing 2). Operation (++) is `string` concatenation. A more general way to define functions in Coq is using pattern matching, as explained in Appendix F.2.

Record instances are non-recursive functions that take no input. The following defines a new instance, *p*1, of type `Person` by invoking the constructor *mkPerson* and supplying first name "`John`" and last name "`Smith`" as arguments:

```
Definition p1 : Person := mkPerson "John" "Smith".
```

Proofs. The body of a function is a proof of termination: *toString* (Listing 2) is guaranteed to terminate and return the evaluation of its body. Here is a definition of *toString* without a body:

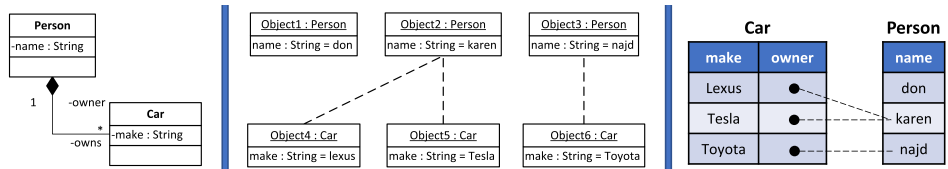
```
Definition toString2 (p: Person) : string.
```

It says that *toString*2 always returns a `string` but does not say *what* `string`. There is no *evidence* that the function terminates for all `Person` inputs. Executing this line lets Coq enter proof mode allowing the user to prove termination. The current state of a proof is shown in a separate panel, which includes information about the goal to be proven and any given facts that typically help in discharging the proof. A goal is usually broken into *subgoals*. The state after the above command is

```
1 subgoal
p : Person
______________________________________(1/1)
string
```

There is only one subgoal. *Hypotheses* are assumptions listed above a horizontal bar; the current subgoal is displayed below. A proof can be discharged in many ways. One can match the subgoal with any term matching the type of subgoal, such as a fixed string, say "`hello`". However, to get the same behavior of the original function *toString*2 is to supply its body in Listing 2:

```
Proof.
apply ((fname p) ++ " " ++ (lname p)).
```



(a) Class Diagram **(b) Object Diagram** **(c) Database Encoding**

Fig. 12. Class diagram, object diagram, and database.

The keyword *Proof* is optional indicating that a proof has started. *apply* is a *tactic* that instructs Coq to match a term against the current subgoal, *string*. Coq offers built-in tactics to reduce current (sub)goals to simpler ones. After executing the last line, the proof state changes to

```
No more subgoals.
```

saying there are no more subgoals to prove. The proof is usually discharged with the keyword *Qed*. However, since *toString*2 is a function definition, *Defined* is used instead. The full proof is

```
Definition toString2 (p: Person): string.
Proof.
apply ((fname p) ++ " " ++ (lname p)).
Defined.
```

In summary, *toString* and *toString*2 are different ways to define the same function: the first in the classical functional way and the last in proof mode.

2.4 Coq Encoding of Relational Databases

Figure 12 shows the *PersonCar* class diagram, an object diagram, and a database of this object diagram. A *table* definition is a Coq record; the table name is the record name and table columns are record fields. Table column types are scalar values, not sets.[2] The definitions of the *Person* and *Car* tables of the *PersonCar* schema are

```
Record Person := mkPerson {        (* Person tuple constructor *)
 name : string;
}.
Record Car := mkCar {              (* Car tuple constructor *)
  make  : string;
  owner : Person;                  (* "foreign" tuple *)
}.
```

A *database* is another Coq record. It contains a list of all T tuples (Coq record instances) for each table T in a schema. A database for *PersonCar* is an instance of

```
Record PersonCar := mkPersonCar {                   (* PersonCar database constructor *)
   pl : list Person;
   cl : list Car;
}.
```

Listing 3 populates tables of `Person` and *Car* with tuples of Figure 12(c) to form the `PersonCar` database, below. This is the encoding of databases used in our proofs.

[2]No set-valued attributes are used in this article; we have proved refactorings in Coq with unnormalized tuples.

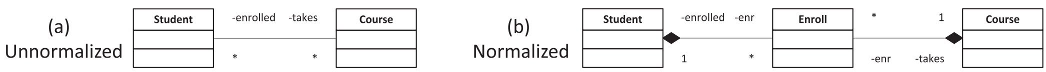


Fig. 13. Unnormalized vs. normalized associations.

```
Definition don        := mkPerson "don".           (* Person tuples *)
Definition karen      := mkPerson "karen".
Definition najd       := mkPerson "najd".
Definition persons    := [don; karen; najd].       (* Person table *)
Definition Lexus      := mkCar "Lexus" karen.      (* Car tuples *)
Definition Tesla      := mkCar "Tesla" karen.
Definition Toyota     := mkCar "Toyota" najd.
Definition cars       := [Lexus; Tesla; Toyota].   (* Car table *)
Definition PersonCarDB := mkPersonCar persons cars. (* database instantiation *)
```

Listing 3. A PersonCar database.

Primary Keys, Tuple IDs, and Associations. Relational tuples have *primary keys*—a subset of columns whose values uniquely identify a tuple. Coq considers two record instances equal if they agree on all field values.

We encoded associations in Listing 3, above. A *Car* tuple has a field *owner* whose value is literally the related Person tuple instead of the tuple's primary key. In the PersonCar database, there are three identical copies of the *karen* tuple: one in the Person table, and two as *owner* values in the *Car* table. This encoding is legal for normalized associations (i.e., * 1◆ and * 0..1◇ associations). For all other associations, the association is normalized, Figures 13(a)⟶(b), by adding an association class *Enroll* with a pair of (1:*) cardinality associations [13, 30, 84].

The Coq encoding of primary keys will *not* handle cyclic databases; details on how to encode such databases are in Appendix C.

Constraints. A database schema lists constraints that its databases must satisfy. To preclude any Person whose name is "Bob" from owning a "Honda" we write

$$\forall c \in Car : c.owner.name = \text{“}Bob\text{”} \Longrightarrow c.make \neq \text{“}Honda\text{”}. \tag{9}$$

Or no Person should own two Teslas:

$$\forall c1, c2 \in Car : (c1.make = \text{“}Tesla\text{”} \wedge c2.make = \text{“}Tesla\text{”}) \Longrightarrow c1.owner \neq c2.owner. \tag{10}$$

Constraints are simply listed after a database's tuple lists, like

```
Record PersonCarWthCons := mkPersonCarWthCons{        (* PersonCarWthCons db constructor *)
  pl : list Person;
  cl : list Car;
  con1: forall c, In c cl → (name (owner c)) = "Bob" → ((make c) <> "Honda");
  con2: forall c1 c2, In c1 cl → In c2 cl → (make c1) = "Tesla" → (make c2) = "Tesla"
  → (owner c1 <> owner c2);
}.
```

As said earlier, constraints make instantiation more complicated. We address this in Section 4.

Association Traversal. Traversing an association from any tuple *c* in table *Car* to its related *owner* tuple *p* in table Person is simple—find the *Car* tuple *c*. The *owner* field yields the associated Person tuple, $p = (owner\ c)$. The dual, going from any Person tuple *p* to its related set of *Car* tuples

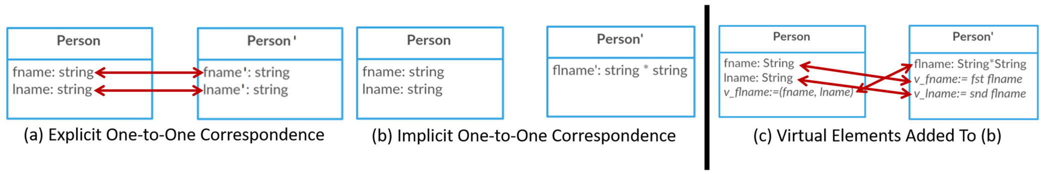


Fig. 14. Invertibility of a refactoring.

requires a function to compute the table of cars `Person` p owns. In database parlance, this is a *right semijoin* [10, 13, 30, 84]. Namely, return cars owned by p and reject others [10]:

```
Definition owns (p:Person) (cl: list Car) := filter (fun c ⇒ p =? (owner c)) cl.
```

Next Sections. We show how to prove round-tripping theorems for *minRefs* without *minCons*, and then with *minCons*. ***Our proofs do not consider the refactoring of OCL constraints***, as this is itself a substantial problem addressed elsewhere [24, 39, 74, 79].

3 PROOFS OF MINIMAL REFACTORINGS WITHOUT MINIMAL CONSTRAINTS

A schema refactoring $\boldsymbol{S}_{\ominus}$ is an invertible co-transformation; not only does $\boldsymbol{S}_{\ominus}$ transform its input database schema $\mathbb{s}$ to output schema $\mathbb{s}$' but also transforms each database instance of $\mathbb{s}$ to a database instance of $\mathbb{s}$', and vice versa. $\mathbb{C}_{\mathbb{T}}$ tells us Equations (7) and (8) are the theorems to prove. A different proof approach is used for each (Sections 3.1 and 3.2). Both use similar steps: encoding structures, defining transformations, and proving invertibility theorems. We use the ***mergeFields***$_{\ominus}$-***splitField***$_{\ominus}$ *minRefs* as exemplars in this section, where *minCons* (e.g., uniqueness and cardinality constraints) are absent in both source and target schemas.

3.1 Database Refactorings

Invertibility is clear when there is a one-to-one correspondence between the domain and *codomain* (output domain) of a refactoring. Figure 14(a) shows each field of `Person` (the domain) has a corresponding field in `Person′` (the codomain). Sometimes this correspondence is hidden or implicit as in Figure 14(b), where a pair of distinct string fields `(fname, lname)` are merged into a single composite string field `(flname′)` as in ***mergeFields***$_{\ominus}$.

To facilitate proofs of invertibility, we make implicit elements—that are explicit in one class diagram but not in the other—explicit by introducing additional classes and fields called ***virtual elements*** (**VEs**) [30, 84], also called *derived elements* in UML [32]. Figure 14(b) is modified to Figure 14(c):

- A virtual full name (v_flname) combines first name ($fname$) and last name ($lname$) into `string` pair in `Person`.
- A virtual first name (v_fname) and a virtual last name (v_lname) in `Person′` returns the first and second elements of $flname$, respectively.

Virtual computations are solely based on data in its database. They do not borrow data from other databases or external data sources. Any number of VEs can be added to a table. Their expressions are defined at the schema level and are (automatically) evaluated at the database level. ***VEs are needed only for database proofs of invertibility and are* not *persistent***.[3]

[3]Underlying each schema is a category. By adding VEs to the domain and co-domain of a *minRef*, we make the categories their schemas isomorphic, a requirement for $\mathbb{C}_{\mathbb{T}}$ equivalence.

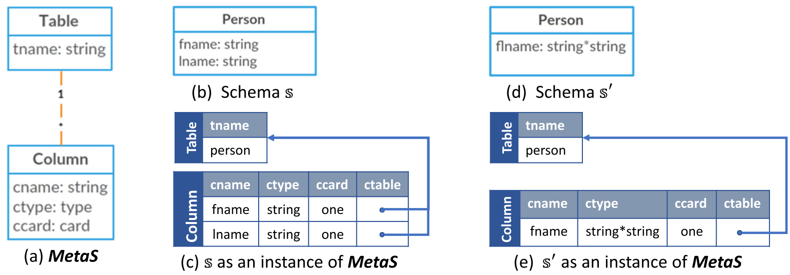


Fig. 15. Schemas are instances of ***MetaS***.

Given the database *minRef* ***mergeFields***$_\ominus$, the tasks in Coq to perform are as follows:

D.1 Declare the minimal source and target schemas of ***mergeFields***$_\ominus$.
D.2 Define the ***mergeFields***$_\ominus$ and ***splitField***$_\ominus$ database *minRef*s.
D.3 State the theorems that ***mergeFields***$_\ominus$ and ***splitField***$_\ominus$ are inverses of each other.
D.4 Prove theorems of Appendix D.3.

The details of task D.τ are presented in Appendix D.τ.

3.2 Schema Refactorings

We now go one level up in the MOF hierarchy and focus on ***mergeFields***$_\ominus$-***splitField***$_\ominus$ *minRef*s at the schema, not database, level. The details are different due to limitations in Coq, but our approach is the same. We define a metaschema to encode database schemas as instances. **Note**: VEs are excluded from schema invertibility proofs as they are needed only for database invertibility proofs.

A Coq Meta-Schema Definition. A schema refactoring cannot be encoded directly as Coq has limited reflection capabilities: i.e., there is no way to access and manipulate Coq record definitions. Therefore, we defined our own metaschema, ***MetaS***, so that every Coq schema (i.e., set of `Record`s) can be encoded as an instance of ***MetaS*** (Figure 15). A ***MetaS*** schema is a pair of tables, literally named *Table* and *Column*, where the name of each Coq record (table definition) t is entered as a row in *Table*, and each column of t is entered as a row in *Column*.

Figure 15(a) is a umlCD of ***MetaS***. Figure 15(c) shows how schema $\mathbb{s}$ of Figure 15(b) is encoded as a ***MetaS*** database: the *Table* has one row for `Person` and the *Column* table has two rows for fields *fname* and *lname*, respectively. Figure 15(d) and (e) are the result of applying ***mergeFields***$_\ominus$ to schema $\mathbb{s}$.

We define *Table* and *Column* as Coq records, and then define a *Schema* as a collection of *Table*s and *Column*s. However, since the type of column might be a reference, not just a primitive type, we define our own generic column type, *CType*. A *CType* cannot be defined as a record since there are different ways to construct it. Instead, we define it *inductively* by allowing multiple constructors separated with a vertical bar "|".[4]

```
Record Table := mkTable {
 tname: string;
};
Inductive CType : Set := String | Bool | Nat      (* primitive Column data types *)
| Pair:  CType → CType →   CType                  (* non-primitive Column data types *)
```

[4]Inductive types do not assign field names, only their types. Field names are defined by (separate) functions. Parameters of each constructor are separated by arrows. The last parameter is the output which is identical to the type being defined.

```
| Option: CType → CType
| List:  CType → CType
| Ref:   Table → CType.
Inductive Card : Set :=
| one | opt1 | many                      (* predefined cardinalities 1,0..1,* *)
| val: nat → Card                        (* custom value, ex 5 *)
| finiteRange: nat → nat → Card          (* a range 4..9 *)
| infiniteRange: nat → Card.             (* a range 3..* *)
Record Column := mkColumn {
 cname : string;
 ctype : CType;                          (* a predefined Column data type *)
 ccard : Card;                           (* a predefined cardinality data type *)
 ctable : Table;                         (* Table row to which this column belongs *)
}.
Record MetaS := mkMetaS {                (* MetaS database constuctor *)
 tbls: list Table;
 cols: list Column;
}.
```

A type can be either a *String*, *Bool*, *Nat*, *Pair* of types, *Option* of a type (which allows nulls), *List* of some type, or a *Reference* to a given table. More variants can be added as needed. Cardinality is defined inductively as well. Possible cardinalities include 1, 0..1, *, a custom value (e.g., 5), and a range between two cardinalities (e.g., 1..5 or 2..*), which correspond to the constructors of *card*.

Schemas 𝕤 and 𝕤' of Figure 15 are databases of ***MetaS***:

```
1  Definition t1 := mkTable "Person".                  (* Person as a Table instance*)
2  Definition c1 := mkColumn "fname" String one t1.    (* fname and lname as Column instances *)
3  Definition c2 := mkColumn "lname" String one t1.    (* both belong to t1, the Person table *)
4  Definition 𝕤 := mkMetaS [t1] [c1; c2].              (* MetaS database for 𝕤 *)
5  Definition c3 := mkColumn "flname" (Pair String String) one t1. (* flname as Col instance *)
6  Definition 𝕤' := mkMetaS [t1] [c3].                 (* MetaS database for 𝕤' *)
```

Listing 4. ***MetaS*** Encoding of Schemas 𝕤 and 𝕤'.

Observe how 𝕤 and 𝕤' are defined (lines 4 and 6). Both schemas have table $t1$. However, 𝕤 has two columns: $c1$ and $c2$, whereas 𝕤' has one column $c3$. This mimics replacing $fname$ and $lname$ with $flname$ when refactoring 𝕤 to 𝕤'.

We proceed as before: given the *minRef* ***mergeFields***$_\ominus$, the tasks in Coq to perform are as follows:

- E.1 Declare the minimal schemas of ***mergeFields***$_\ominus$-***splitField***$_\ominus$ in ***MetaS***.
- E.2 Define the ***mergeFields***$_\ominus$ and ***splitField***$_\ominus$ schema refactorings.
- E.3 State their round-tripping theorems and proofs.

As before, the details of task E.τ are presented in Appendix E.τ. Appendices E.1 and E.2 are straightforward; the proof in Appendix E.3 is tedious and non-trivial.

4 PROOFS OF MINIMAL REFACTORINGS WITH MINIMAL CONSTRAINTS

Coq proofs for *minRefs* with *minCons* are more complex than without, due in part to Coq following *Intuitionistic Logic* [97], not classical logic [96]. These difficulties are explained next.

4.1 Proposition Complexities of Intuitionistic Logic

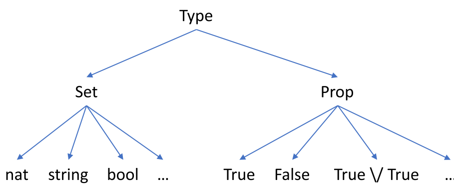


Fig. 16. Coq type hierarchy.

Coq propositions are not expressions to be evaluated but are types that belong to `Prop`. Familiar types like *nat* and *bool* belong to `Set`. Figure 16 depicts Coq's type hierarchy. `bool` expressions (with operators like `&&`, `||`, and `=?`) belong to the computational universe of Coq and can be evaluated to either $true$ or $false$. On the other hand, *Prop* expressions (with operators like `/\`, `\/`, and `=`) *cannot* be evaluated but may only be *proven*. For example, the Boolean expression `true || true` evaluates, in Coq, to `true`. However, the corresponding propositional expression `True\/True` does *not* evaluate to $True$. Instead, one must use (inference) rules, provided in a Coq library, to show that `True\/True` reduces to $True$.

A proposition may be proven in different ways. Each proof is called a *proof object*. The type `nat`, when considered as a proposition, has every natural number as an evidence or proof. The theorem p below is discharged by selecting number 2 as a proof:

```
Theorem p : nat.
Proof. exact 2. Qed.
```

Another proof would use a different number, say 3:

```
Theorem q : nat.
Proof. exact 3. Qed.
```

Here p and q represent two identical theorems of the same type with different proofs. Our instinct says p and q represent different proofs of the same theorem, and intuitively should be equal. After all, we don't care how a theorem is proven. Our main interest is knowing if the theorem holds. This line of thinking relies on a known mathematical axiom called *proof irrelevance* [21]: any two objects of the same proposition are equal.

Coq proceeds differently: different proofs *are* different objects, and thus $p \neq q$. Coq does not have the proof irrelevance axiom as part of its theory and consequently must be told explicitly when to apply `proof_irrelevance`. So if we want Coq to consider p and q equivalent, we must write

```
Theorem th: p = q.
Proof.
apply proof_irrelevance.
Qed.
```

Another complexity in Coq is showing the equivalence of two instances of the same structure *with* constraints. One would think that equivalence is established just by showing the values of corresponding fields are identical. Not so. Consider the following definition of positive numbers [66]. The structure involves a field *val* of type *nat* and a constraint $val > 0$:

```
Record PositiveNum := mkNum {
  val: nat;
  is_pos: val > 0;
}.
```

Two instances of $PositiveNum$, a and b, are equal if (1) $(val\ \ a) = (val\ \ b)$ and (2) $(is\_pos\ \ a) = (is\_pos\ \ b)$. The first requirement is straightforward, but the second is not. Typically, if we know that $(val\ \ a) = (val\ \ b)$, and $(is\_pos\ \ a)$ holds, we would conclude that $(is\_pos\ \ b)$ must also hold. However, Coq cannot do this inference: $(is\_pos\ \ a)$ and $(is\_pos\ \ b)$ are *different* types ($val\ \ a > 0$ and *val b* > 0, respectively), and therefore are not equal. To solve this, we need to *transport* (*is_pos a*) from a proof of type (*val a* > 0) to a proof of type (*val b* > 0) to prove (*is_pos b*).

Transport Example 1. The following theorem attempts to prove that if the values of two *PositiveNum*s are equal, then they are equal (by saying nothing about propositions).

```
1    Theorem A_equals_B (A B : PositiveNum):
2    (val A = val B) → (A = B).
3    Proof.
4    ...                 (* some script here *)
```

```
1 subgoal
m : nat
g : m > 0
n : nat
h : n > 0
p : m = n
_______________________________________(1/1)
{| val := m; is_pos := g |} =
{| val := n; is_pos := h |}
```

```
5    f_equal.           (* does nothing because g and h have different types *)
6    Abort.             (* quit the proof *)
```

The tactic used in Line 5, **`f_equal`**, matches corresponding fields. It fails because the proof objects, g and h, are of different types (`m > 0` and `n > 0`, respectively).

Transport Example 2. Let `p` be the proof object of the equality statement `m = n`, which can be used to transport instances of type `m > 0` to instances of type `n > 0`. We do this by proving the lemma, which we call `transport`. (The *transport* concept is part of Coq's foundation but is ***not*** a keyword of Coq).

```
Lemma transport (x y : nat) (H: x = y) (G: x > 0): y > 0.
```

transport takes as input two numbers `x` and `y`, an evidence H stating that x and y are equal, and another evidence G stating that `x > 0`, and outputs a proof object of type `y > 0`. In this case, G is the proof object to transport along H. The proof of the lemma is trivial: H is used to rewrite G where occurrences of x are replaced with y in G. The result matches the goal ($y > 0$) which concludes the proof:

```
Lemma transport (x y : nat) (H: x = y) (G: x > 0): y > 0.
Proof.
 rewrite H in G.
 assumption.
Qed.
```

With *transport*, we can prove theorem `A_equals_B` but must state it as

```
Theorem A_equals_Bv2 (A B : PositiveNum) (p: val A = val B)
(q: (transport (val A) (val B) p (is_pos A)) = (is_pos B) ):
A = B.
Proof.
...        (* some script here *)
f_equal.  (* it works! *)
Qed.
```

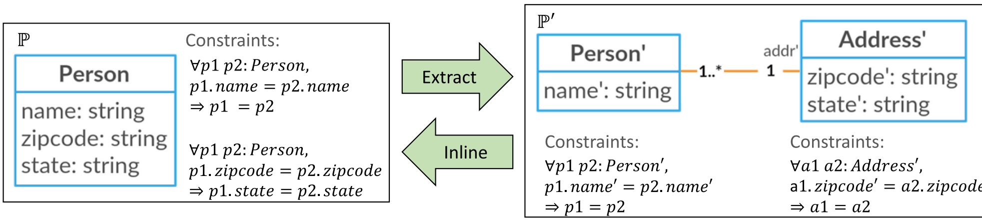


Fig. 17. Extract/inline class refactoring.

The theorem says, given
- ***two instances*** A ***and*** B ***of*** $PositiveNum$***,***
- ***a proof that*** A ***and*** B ***share the same value for field*** val***, and***
- ***a proof that the proof objects*** $(is\_pos\ A)$ ***and*** $(is\_pos\ B)$ ***are equal under transportation,***

then A ***and*** B ***are equal.***

4.2 Database Refactorings with Minimal Constraints

Metamodel $\mathbb{P}$ in Figure 17 has one class, `Person`. Each person has a `name`, resides in a `zipcode` and in a `state`. A constraint of $\mathbb{P}$ is *uniqueness*: `name` is the primary key of `Person`:

$$\forall p1, p2 \in Person : p1.name = p2.name \implies p1 = p2.$$

Also, a `zipcode` belongs to only one `state`. So, 78704 cannot be both a zipcode in Texas and, say, California. This is captured by the `sameState` constraint:

$$\forall p1, p2 \in Person : p1.zipcode = p2.zipcode \implies p1.state = p2.state.$$

The refactored metamodel, $\mathbb{P}'$, has two classes: `Person′` and `Address′` where the residence information (i.e., `zipcode` and `state`) is extracted from `Person` into a newly created class `Address′`. Now, each person from `Person′` has ***one*** `Address′`, and each address hosts ***at least one*** `Person′`. We call this $_{min}Ref$ ***extract***$_\ominus$-***inline***$_\ominus$. (In database parlance, it is called *table normalization*).

Observe that if the primary key constraint was removed or if the cardinalities were chosen differently, the refactoring would be incorrect, leading to data inconsistencies. We call such constraints *minimal*.

Given the above, the tasks in Coq to perform are as follows:
- F.1 Declare the target schemas $\mathbb{P}$ and $\mathbb{P}'$ in Coq.
- F.2 Define the ***extract***$_\ominus$ and ***inline***$_\ominus$ database $_{min}Ref$s;.
- F.3 State their round-tripping theorems and proofs.

The details of task F.τ are presented in Appendix F.τ. None of these tasks are trivial.

4.3 Schema Refactorings with Minimal Constraints

The correctness of a $_{min}Ref$ at the database level was shown in Appendix F. At the schema level, the focus is on structural and syntactic details. As we are working with a concrete minimal schema, field names, table names, and constraint expressions are fixed (prespecified terms).

Recall our metaschema ***MetaS*** is a list of tables and columns (Section 3.2). It now must be extended to accommodate $_{min}Con$s. The first challenge is: in what language are $_{min}Con$s expressed? And then how to recognize if a schema satisfies a $_{min}Con$, as even simple constraints can be written in different ways, like the XOR of predicates P and Q:

$$(P \wedge \neg Q) \vee (\neg P \wedge Q) \quad \text{OR} \quad (P \vee Q) \wedge (\neg P \vee \neg Q) \quad \text{OR} \quad Q \vee P \Rightarrow \neg(Q \wedge P) \quad \text{OR} \quad \dots .$$

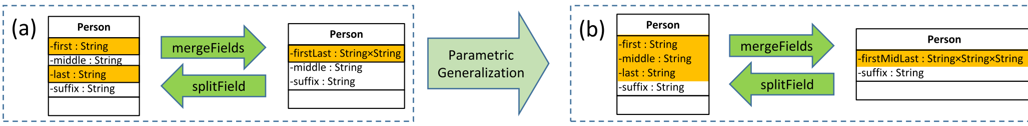


Fig. 18. ***pullUp***$_\ominus$-***pushDown***$_\ominus$ *minRefs* parametrically generalized to *parRefs*.

If OCL was used to declare an *minCons*, it would be daunting to take $k \geq 1$ OCL constraints and deduce if a minimal constraint holds. A much simpler solution, which we adopt, is to have a special single syntax (or term) for each *minCon* as we expect few distinct *minCon* types; most are related to cardinality and tuple uniqueness. Doing so enables an MDE refactoring engine to quickly determine if a particular set of *minCons* holds.

Consider the *minCons* of `Schema`$\mathbb{P}'$: *personKey*$'$, *addressKey*$'$, and *card*$'$. All can be expressed in a uniform way using predefined general-purpose constraints: `key`, `funDep`, and `nonNull`. `key(X)` declares a non-empty set X of fields (with non-null values) to be a primary key of a designated table. `funDep(X,Y)` defines a functional dependency $X \rightarrow Y$ where X and Y are non-empty disjoint sets of fields of the same table [30, 84]. And `notNull(F)` declares a field F never to have a null value. Therefore, instead of having to write *personKey*$'$ as

$$\forall p1, p2 \in Person' : p1.name' = p2.name' \Rightarrow p1 = p2,$$

it can be stated briefly:

$$Person'.key(\{name\}).$$

Using special syntax for *minCons* (a) makes it easy for a refactoring engine to check if a specific *minCon* holds and (b) simplifies the writing of schema refactorings. As we are working with concrete minimal schemas, *minCons* can be hard-coded as `strings`. The constraints are as follows:

```
Definition personKey   := "Person.key({name})".       (* Person Constraints *)
Definition sameState   := "Person.funDep({zipcode},{state})".

Definition personKey'  := "Person'.key({name'})".     (* Person' Constraints *)
Definition card'       := "Person'.nonNull(addr')".

Definition addressKey' := "Address'.key({zipcode'})". (* Address' Constraint *)
```

The metamodel $\mathbb{P}$-to-$\mathbb{P}'$ refactoring translates *minCons* `personKey` to `personKey'`, `sameState` to `addressKey'`, and `personKey`$\wedge$`sameState` to `card'`. The $\mathbb{P}'$-to-$\mathbb{P}$ refactoring restores `personKey` and `sameState`. The proofs of invertibility are trivial. Simplicity is due to the fact that we are looking at a concrete instance and that *minCons* are recognizable strings. When a refactoring is elevated to its generalized form, ***MetaS*** would require an additional table (list) of *minCons*. The proof would be a bit more involved and is left for future work.

5 REMAINING STEPS AND OTHER FUTURE WORK

5.1 Parametric Generalizations ($R_\oplus$)

mergeFields$_\ominus$ merges two `String` fields into a `Pair<String>` field (Figure 18(a)). How could this *minRef* be generalized?

One way would be to merge $n > 2$ fields; Figure 18(b) illustrates $n = 3$. Another way would replace the `String` parameter of `Pair<>` with a different type (e.g., `Integer`). Both are examples of *parametric generalizations*, where ***mergeFields***$_\ominus$ is given more arguments to become a *parametric refactoring* (*parRef*), ***mergeFields***$_\oplus$, denoted by $\oplus$.

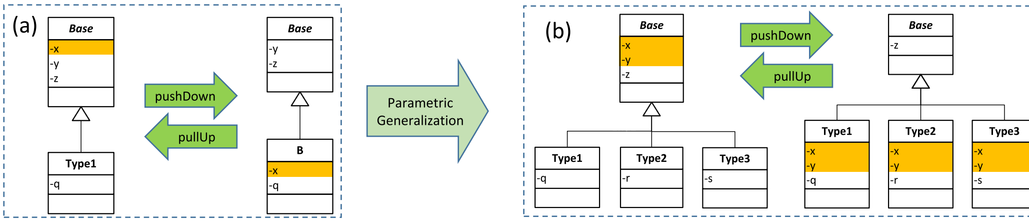


Fig. 19. ***mergeFields***$_\ominus$-***splitField***$_\ominus$ *minRefs* parametrically generalized to *parRefs*.

Similarly, ***pushDown***$_\ominus$ pushes down one field of an abstract class to its lone subclass (Figure 19(a)). A parametric generalization pushes down multiple (≥2) fields together; another allows multiple (≥2) subclasses (Figure 19(d)). Combinations of generalizations are to be expected.

Parametric generalizations make small changes to the Coq definition of a *minRef*, typically by adding loops or using arguments for previously fixed values.

Parametric generalizations enlarge the domain and co-domain of a *minRef* $\boldsymbol{R}_\ominus{:}\{cd\}{\rightarrow}\{cd'\}$ to a *parRef* $\boldsymbol{R}_\oplus{:}\Omega_\oplus{\rightarrow}\Omega'_\oplus$, with a larger domain ($\Omega_\oplus$) and co-domain ($\Omega'_\oplus$), Figure 20(a)⟶(b):

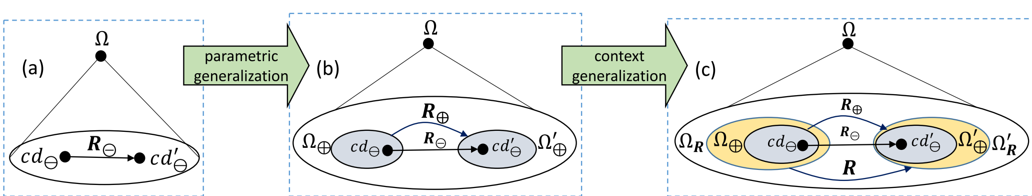


Fig. 20. Effects of a parametric and context generalizations.

and *parRef* round-tripping theorems add another level of quantification. Equations (3) and (4) become

$$\Big(\forall cd\in\Omega_\oplus\ :\ cd=\boldsymbol{R}_\oplus^{-1}(\boldsymbol{R}_\oplus(cd))\Big)\quad\bigwedge\quad\Big(\forall cd'\in\Omega'_\oplus\ :\ cd'=\boldsymbol{R}_\oplus(\boldsymbol{R}_\oplus^{-1}(cd'))\Big),\tag{11}$$

$$\Big(\forall cd\in\Omega_\oplus,\ \forall d\in cd\ :\ d=\boldsymbol{R}_\oplus^{-1}(\boldsymbol{R}_\oplus(d))\Big)\quad\bigwedge\quad\Big(\forall cd'\in\Omega'_\oplus,\ \forall d'\in cd'\ :\ d'=\boldsymbol{R}_\oplus(\boldsymbol{R}_\oplus^{-1}(d'))\Big).\tag{12}$$

5.2 Contextual Embeddings and Full Refactorings (*R*)

A refactoring engine offers its users *full* refactorings, where a refactoring target umlCD T is embedded in a larger umlCD C, written $T{\hookrightarrow}C$. C is the umlCD of the user's MDE metamodel, called a *context*. Unlike prior sections, T has class, field, and association names that are expected to be *different* from those hardwired in a *parRef* definition.

In Figure 21, full refactorings ***mergeFields-splitField*** are applied to a particular class (*Dog*) that is embedded in a larger umlCD (a context). This is accomplished by extending ***mergeFields***$_\oplus$-***splitField***$_\oplus$ with additional parameters for each class, field, and association name (to make name bindings general), focusing on the class(es) to transform, and leaving the remaining diagram intact (see [76, 77] for details). Extending proofs of $\boldsymbol{R}_\oplus$ to $\boldsymbol{R}$ requires yet another proof elaboration.

A context generalization enlarges a *parRef* $\boldsymbol{R}_\oplus{:}\Omega_\oplus{\rightarrow}\Omega'_\oplus$ to express a full umlCD refactoring $\boldsymbol{R}{:}\Omega_{\boldsymbol{R}}{\rightarrow}\Omega'_{\boldsymbol{R}}$ with its expected broad domain and co-domain, Figure 20(b)⟶(c).

Round-tripping theorems for full refactorings are generalizations of *parRef* theorems, Equations (11) and (12), with a broader scope of quantification, i.e., $\Omega_\oplus$ is widened to $\Omega_{\boldsymbol{R}}$ and $\Omega'_\oplus$ is widened to $\Omega'_{\boldsymbol{R}}$. $\Omega_{\boldsymbol{R}}$ is the subdomain of Ω (containing all class diagrams) that

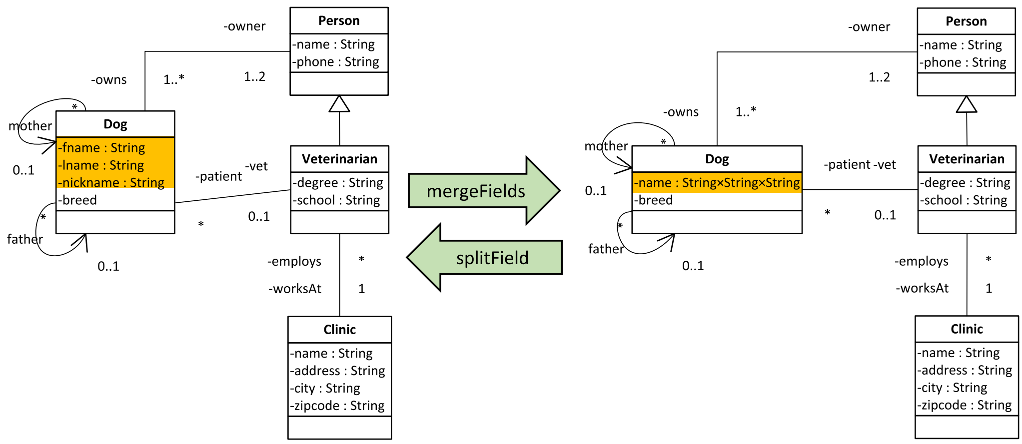


Fig. 21. Applying ***mergeField*-*splitFields*** in context umlCDs.

satisfies the preconditions of $\boldsymbol{R}$; $\Omega'_{\boldsymbol{R}}$ is the subdomain of Ω that satisfies the postconditions of $\boldsymbol{R}$. Equations (11) and (12) become

$$\Big(\forall cd \in \Omega_{\boldsymbol{R}} \ : \ cd = \boldsymbol{R}_{\oplus}^{-1}(\boldsymbol{R}_{\oplus}(cd))\Big) \quad \bigwedge \quad \Big(\forall cd' \in \Omega'_{\boldsymbol{R}} \ : \ cd' = \boldsymbol{R}_{\oplus}(\boldsymbol{R}_{\oplus}^{-1}(cd'))\Big), \tag{13}$$

$$\Big(\forall cd \in \Omega_{\boldsymbol{R}}, \forall d \in cd \ : \ d = \boldsymbol{R}_{\oplus}^{-1}(\boldsymbol{R}_{\oplus}(d))\Big) \quad \bigwedge \quad \Big(\forall cd' \in \Omega'_{\boldsymbol{R}}, \forall d' \in cd' \ : \ d' = \boldsymbol{R}_{\oplus}(\boldsymbol{R}_{\oplus}^{-1}(d'))\Big). \tag{14}$$

As said earlier, we found tackling the most generalized refactoring possible—namely, minimal refactorings with parametrization and context generalizations—was too daunting. Instead, start with a *minRef* proof and incrementally extending it would be more understandable, doable, and easier to explain, as the scope of each task is smaller.

5.3 A Sketch of a Relational Algebra Theorem Prover

For some time, we suspected that Coq was not the right prover to use. A prover that verified *Relational Algebra* ($\mathbb{R}_{\mathbb{A}}$) identities, in our opinion, would have been better. We found leads [15, 17, 27] but no usable tools, so we continued with Coq.

Referees of this article brought ***Database Model Management*** (**DbMM**) [18, 19, 60, 72, 73] to our attention. DbMM is the counterpart to work on MDE Model Management—propagation of changes to a metamodel and its models—where refactorings are special cases. DbMM uses $\mathbb{R}_{\mathbb{A}}$ to specify and analyze changes to *both* relational schemas and their databases. This literature supported our intuitions that Coq abstractions and specifications were too low-level. We sketch and explain below why a prover based on $\mathbb{R}_{\mathbb{A}}$ might be better.

Relational Algebra. Consider this $\mathbb{R}_{\mathbb{A}}$ expression that joins tables R and S and then projects fields $R.A$ and $S.B$ [30, 84]:

$$RS \ = \ \Pi_{R.A,\ S.B}\,(R \bowtie S). \tag{15}$$

Observe that projection (Π), natural join ($\bowtie$), and indeed all $\mathbb{R}_{\mathbb{A}}$ operations are co-transformations. That is, each $\mathbb{R}_{\mathbb{A}}$ operation encodes a pair of operations: one on schemas and another on tables. This unification leads to a single and compact specification for round-tripping schema *and* database refactorings. To show this, we mix Coq-like notations with $\mathbb{R}_{\mathbb{A}}$ expressions to recast the theorems of Sections 3 and 4. The end result has a flavor of Algebraic Specifications [17, 80, 85]. We use four $\mathbb{R}_{\mathbb{A}}$ operations; the first three are standard [30, 84, 90]:

- Projecting columns $c_1, c_2, \ldots,$ from table T to produce table $\widehat{T}$: $\Pi_{c_1, c_2, \ldots,} T = \widehat{T}$.
- Projecting named columns $n_1, n_2, \ldots,$ whose original column names are $c_1, c_2, \ldots,$ from table T to produce table $\widehat{T}$: $\Pi_{n_1:c_1,\ n_2:c_2,\ \ldots,} T = \widehat{T}$.
- Natural join of tables R and S to produce table $\widehat{RS}$ [30, 84]: $R \bowtie S = \widehat{RS}$.
- **Database Constructor:** Let schema $\mathbb{P}$ have two tables $\{R, S\}$. A new instance d of $\mathbb{P}$, whose table expressions are $\{Rx, Sx\}$, is formed by: $d = \mathbb{P}[Rx, Sx]$.

mergeFields$_\ominus$***-splitField***$_\ominus$**.** As in our Coq proof, a pair of axioms is used (Figure 22). Equation (16) states the fst of a $Pair(a, b)$ is a, and Equation (17) states the snd of that Pair is b:

$$fst(Pair(a, b)) = a, \quad (16)$$
$$snd(Pair(a, b)) = b. \quad (17)$$

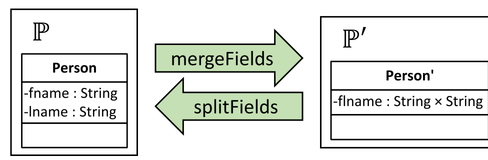


Fig. 22. ***mergeFields***$_\ominus$***-splitField***$_\ominus$.

Some helper functions are needed. Equation (18) translates a $Person$ table to a $Person'$ table and Equation (19) is its inverse:

$$\texttt{Definition } toPerson'(p : Person) : Person' := \Pi_{flname:Pair(fname,\ lname)}(p). \quad (18)$$

$$\texttt{Definition } toPerson(p' : Person') : Person := \Pi_{fname:fst(flname),\ lname:snd(flname)}(p'). \quad (19)$$

Equation (18) projects $Person$ table p to a $Person'$ table whose $flname$ column has $Pair$ $(fname, lname)$ values. Equation (19) projects $Person'$ table p' to a $Person$ table whose columns $fname$, $lname$ have values $fst(flname)$ and $snd(flname)$.

The round-tripping theorems are essentially identical to Equations (5) and (6) (below) and can be proven manually using known $\mathbb{R}_\mathbb{A}$ identities and Equations (16) and (19).

$$\texttt{Theorem P\_roundTrip:} \quad \forall\, d\ : \mathbb{P}, \quad d\ = \mathbb{P}\left[\, toPerson(toPerson'(d.Person))\,\right]. \quad (20)$$

$$\texttt{Theorem P'\_RoundTrip:} \quad \forall\, d'\ : \mathbb{P}', \quad d' = \mathbb{P}'\left[\, toPerson'(toPerson(d'.Person'))\,\right]. \quad (21)$$

extract$_\ominus$***-inline***$_\ominus$**.** Three helper functions are needed. Equation (22) produces a $Person'$ table from a $Person$ table by projecting the $name$ and $zipcode$ columns. (The $Person$ table has two columns: $name$ and $zipcode$; $zipcode$ implements the $Person' \overset{1}{\text{—}} Address'$ association of Figure 23). Equation (23) produces an $Address'$ table from a $Person$ table by projecting the $zipcode$, $state$ columns. Equation (24) reconstructs a $Person$ table by a natural join of the $Person'$ and $Address'$ tables:

$$\texttt{Definition } toPerson'(p : Person) : Person' := \Pi_{name,\ zipcode}(p). \quad (22)$$

$$\texttt{Definition } toAddress'(p : Person) : Address' := \Pi_{zipcode,\ state}(p). \quad (23)$$

$$\texttt{Definition } toPerson(p' : Person', a' : Address') : Person := p' \bowtie a'. \quad (24)$$

Figure 23 shows both umlCDs, $\mathbb{P}$ and $\mathbb{P}'$, with their constraints. $\mathbb{P}$ has two functional dependencies that permit the partitioning of $Person$ into $Person' \overset{1..*\quad 1}{\text{———}} Address'$. $\mathbb{P}'$ retains these dependencies and adds two more constraints. The $Person'$ table can be reconstructed from ($Person'$ $\bowtie$

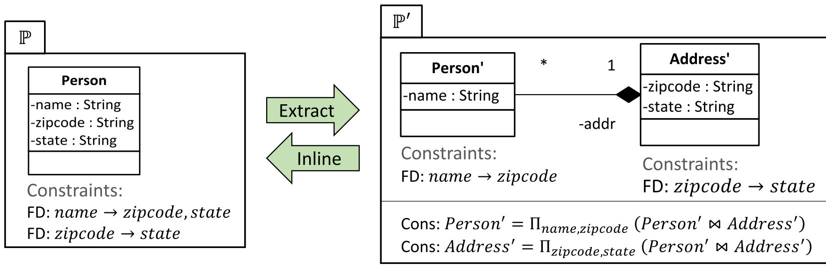


Fig. 23. ***extract***$_\ominus$-***inline***$_\ominus$.

Address′) followed by a projection of the *name*, *zipcode* columns. This constraint means that every *Person′* tuple joins with one *Address′* tuple. That is, it encodes the 1 cardinality of association $Person' \overset{1}{\text{——}} Address'$. The second constraint encodes the 1..* cardinality of association $Person' \overset{1..*}{\text{——}} Address'$.

As before, the round-tripping theorems are essentially identical to those used in Appendix F.3:

$$\texttt{Definition}\ p'(d:\mathbb{P}):\mathbb{P}' = \mathbb{P}'[\ toPerson'(d.Person),\ toAddress'(d.Person)\ ]. \tag{25}$$

$$\texttt{Definition}\ p(d':\mathbb{P}'):\mathbb{P} = \mathbb{P}[\ toPerson(p'.Person,\ p'.Address')\ ]. \tag{26}$$

$$\texttt{Theorem P\_roundTrip:}\ \ \forall\, d\ :\mathbb{P},\quad d\ =\ p(\ p'(\ d\ )\ ). \tag{27}$$

$$\texttt{Theorem P'\_RoundTrip:}\ \ \forall\, d':\mathbb{P}',\ d'\ =\ p'(\ p(\ d'\ )\ ). \tag{28}$$

Recap. Coq specifications of umlCD refactorings are too low-level; $\mathbb{R}_{\mathbb{A}}$ specifications are more appropriate as they are at the right level of abstraction, namely, as $\mathbb{R}_{\mathbb{A}}$ co-transformations. Consider the *minCons* of $\mathbb{P}'$: that both *Person′* and *Address′* tables can be recovered from their join. This precondition is admittedly not obvious from our proof in Section 4, but is a non-trivial and precise precondition of ***inline***$_\ominus$. Any *Person′* tuple that references a non-existent *zipcode* in the *Address′* table, or any *zipcode* in the *Address′* table that is not referenced by a *Person′* tuple will violate the preconditions of the ***inline***$_\ominus$ refactoring.

6 RELATED WORK

6.1 MDE Refactorings

The work of Gheyi et al. [35, 58] was very influential to us. These were the earliest papers to our knowledge that used a theorem prover, ***Prototype Verification System*** (**PVS**), to verify the correctness of umlCD refactorings. Refactorings were defined between Alloy modules [42]. (**Note:** The term *model* is standard for an Alloy specification; we replaced it with *module* to avoid confusion with MDE terminology). They argued that refactorings can be analyzed by translating before-and-after umlCDs to Alloy and proving umlCD equivalence. Two Alloy modules are said to be *semantically equivalent* if their corresponding set of instances are identical. Correspondence is achieved through mappings, or *views* a.k.a. virtual elements, which may involve a computation to recover a missing field or association in the target class diagram. As their views only find corresponding fields and associations, and not classes, their definition could not be used to prove intuitively equivalent modules (which the authors acknowledge) [35]. For this reason, in some cases they could only prove (one-way) embeddings in place of refactorings. We were able to take their example and prove bi-directional embeddings [2].

6.2 Category Theory

Using $\mathbb{C}_{\mathbb{T}}$ as a foundation to study and formalize refactorings is not new [48, 79, 82, 91]; it is our holistic use of $\mathbb{C}_{\mathbb{T}}$ that is novel.

Schulz et al. [82] studied metamodel refactorings using $\mathbb{C}_{\mathbb{T}}$. Horn clauses expressed metamodel constraints. They, like us, asserted "*refactorings preserved model data*," but the inverse of a refactoring is itself a refactoring was not explored. "Refactorings" were thus embeddings, not equivalences, which allows for many more transformations to be called "refactorings" than we would accept.

6.3 Unbounded-Level MOFs

There is research in MDE that removes bounds on three-level MOFs, where n-level MOFs ($n > 3$) are possible [50]. Our work focuses on the classical case of a fixed meta-metamodel at level `n = 3` and co-refactorings at the metamodel level, $n = 2$, and model level, $n = 1$. $\mathbb{C}_{\mathbb{T}}$ suggests what a refactoring at level $n > 3$ means. An *n-level refactoring* is a level-recursive co-transformation. The initial refactoring is applied to a model at level `n-1`. Its instances at level `n-2` are co-refactored. Affected instances at level `n-3` are then co-refactored, recursively until terminating at the model level, `n = 1`. Without examples, this is hard to imagine, although it is indeed reasonable.

6.4 Refactoring Verification

Different techniques were developed to reason about model and/or metamodel refactorings.

Maoz et al. [54] analyzed the correctness of class diagram refactorings using the Alloy Analyzer. They deeply embed class diagrams in Alloy to compare and manipulate two or more umlCDs in one Alloy module. Due to the nature of Alloy, the scope of analysis is restricted, thus equivalences can only be proven up to some bound.

In another work [55], the same authors computed the semantic differences between two class diagrams. Alloy was used to encode the source and target umlCDs in an Alloy module making it possible to instantiate the module to reveal semantic differences. That is, each instance corresponds to an object diagram that is valid for the source umlCD but not the target.

Costa et al. [26] used Prolog to reason about differences in a pair of umlCDs using a base umlCD as a common ancestor. These umlCDs are translated to Prolog facts and then each of the two versions is compared against the base. The set of changes from the first version is compared to those from the second. By following a set of semantic rules, e.g., an abstract class is equivalent to an interface if they have the same name and same elements and if all the methods in the abstract class are defined as abstract, a conclusion is then derived: (1) first and second umlCDs are equivalent, (2) one includes the other, or (3) they are in conflict.

MDE Model Management. Sträten et al. [88] discussed model refactorings in terms of behavioral properties. The behavior of a model is captured through state machine and sequence diagrams. Both representations must be *consistent*, i.e., the same call sequences must be present in both diagrams. When a model is modified, its behavior is updated accordingly such that consistency is preserved. Moreover, in a model refactoring, *call preservation* must also be satisfied, i.e., the same call sequence is invoked on the original and refactored model. The emphasis is on preserving the sequence itself, not its evaluation. The authors formalized consistency and preservation properties, and verified these properties hold using *Description Logic*. A supporting prototype tool was also developed. Although it is important to ensure such properties, our definition of model refactorings is based on data, not behavior: call preservation does not guarantee matching results if data is not preserved.

Sultana and Thompson [89] explored transformations (refactorings and extensions) of Haskell programs with proofs of correctness using the Isabelle/HOL proof assistant. Proving programs correct, even Haskell programs, is far more difficult than refactoring MDE class diagrams and OCL constraints in our opinion. Further, the inverse of a refactoring is itself a refactoring was not explored. They did consider "lifting," which is an equivalence. But other "refactorings" included extensions to types—adding new operations, which are not equivalences but embeddings or edits by our definition. (Hint: categories without an arrow (operation) are not equivalent to categories with a new arrow (operation)).

6.5 Co-Transformations

Refactorings are *co-transformations* where models are updated whenever their metamodels are transformed to preserve conformance. More refined ideas occur under different topics as well, including co-evolution and co-adaption [93]. For example, König et al. [48] presented a framework based on $\mathbb{C}_{\mathbb{T}}$ and triple-graph-grammars to auto-generate transformations at the instance level w.r.t. a transformation at the metamodel level. We used a bit less $\mathbb{C}_{\mathbb{T}}$ in our article to achieve a similar but more restrictive result on refactorings.

MDE Model Management. Herrmannsdörfer et al. [41] introduced COPE, an approach and tool to help manually migrate models whenever their corresponding metamodel evolves. Like our work, they predefine a set of reusable co-transformations: a pair of [metamodel adaptation and its corresponding model migration]. After a co-transformation takes place, metamodel consistency (i.e., satisfying the meta-metamodel constraints) and model conformance must be checked. This differs from our approach where transformations are certified (by a theorem prover) to produce correct results. It is not clear if or how OCL constraints are handled.

A theoretical model to facilitate the migration of data of an evolved metamodel was developed by Täntzer et al. [91]. $\mathbb{C}_{\mathbb{T}}$ was used whose interpretation was grounded in algebraic graph transformations. Refactorings were not explicitly considered as they are a special case of graph transformations. The approach was realized by a tool [53] showing in detail how graph transformations form a theoretical basis for MDE co-transformations. The correctness of transformations, refactorings included, was not a focus of their work.

Berg and Yu [16, 100], addressed the problem of re-establishing consistency of models after performing a metamodel refactoring. They present a formal framework to define transformation rules for each metamodel refactoring. They argue that rules can be used to develop an analysis engine that (1) derives corresponding model transformations (by analyzing the effects of applying the rules); and (2) automatically detect candidate refactorings. Implementing analysis engines was left for future work.

6.6 Transformation Verification

To verify the correctness of an MDE transformation, various tools have been used.

Anastasakis et al. [4] used Alloy to specify source and target metamodels in addition to a set transformation rules. If Alloy was unable to simulate a transformation, this indicated that the transformation rules were inconsistent.

Berramla et al. [20] used Coq to prove the correctness of an algorithm that transforms a given state diagram to its corresponding Petri Net representation.

Calegari et al. [25] presented a general framework in Coq that can be used to prove the correctness of model transformation with respect to a target metamodel and transformation rules. A transformation is correct if it meets its specification. Their work is similar to ours where metamodels are directly encoded (using records and inductive types). However, inheritance is not fully captured as it is represented merely as an association without enforcing its semantics with supporting constraints. Another difference is that their transformations are declarative (i.e., specified through propositions), whereas ours are imperative (i.e., defined by means of functions). Finally, we go beyond this by showing the *invertibility* property of refactorings to establish semantic equivalence (i.e., data preservation) as opposed to only verifying a transformation guarantees conformance preservation.

MDE Model Management. Ledang and Dubois [51] proved model transformations using the B formalism where B provers were used for analysis and proofs. Based on their verification technique, a transformation is guaranteed to respect its predefined invariants and to produce models that conform to the target metamodel. Although their approach guarantees that a transformation meets its specification (via invariants and conformance rules) their work does not guarantee that the defined invariants preserve data.

Bi-directional transformations preserve consistency between source and target models. Ehrig et al. [29] formalized bi-directional transformations using $\mathbb{C}_{\mathbb{T}}$ and triple graph grammars, and showed that these transformations are *information-preserving* between related graphs. Only *common* information between models is preserved as opposed to all data—a basic requirement in refactorings.

6.7 Database Refactorings

Among the earliest works on umlCD-like refactorings, circa 1982 before MDE was recognized as a discipline, is in the database literature. Atzeni et al. [6] defined schema equivalence in terms of queries and functional dependencies, and manually proved properties that implied equivalence.

A recent and impressive contribution is by Wang et al. [94]. They automatically verified the equivalence of database-driven applications before and after a database refactoring. Equivalence was based on queries: evaluating corresponding queries on source and target schemas must always return the same data. This is done by relating database states through a *bisimulation invariant*. Such an invariant must be sufficient (i.e., covers all queries from interfacing applications) and inductive (i.e., always holds). They developed a tool that generates a possibly suitable invariant and attempt to automatically prove its correctness using the Z3 SMT solver. As our focus is primarily concerned with schemas rather than applications interfacing them, our equivalence definition is more restrictive: not only queries defined by the application must yield the same result, but *any* possible set of corresponding queries. Stated differently, if data isn't preserved, the output of some query that was not considered will be different in the source and target schemas.

DbMM. Bernstein and colleagues had a series of papers circa 2007 that (in our opinion) revolutionized DbMM [18, 19, 61, 72, 73]. DbMM is a generic approach to deal with schema updates, their impact on schema instances (databases), and application queries + constraints. Today, database schemas can be expressed in an astonishing number of ways, including non-standard schema declarations in different relational DBMSs, XML schemas, ER-schemas, and OO languages (Java, .Net) [65]. Further, different query languages (SQL, XQuery, XSLT, ER-SQL) give rise to a large universe of complex translations. DbMM not only executes query mappings, it also propagates updates, notifications, exceptions, access control fights, and provenance.

Although $\mathbb{C}_{\mathbb{T}}$ is not used as a foundation for DbMM, it certainly could be as the main technical ideas of DbMM are transformations (i.e., total functions) and their compositions.

Refactorings are special cases of schema updates. DbMM opens up a more general vision of refactorings. In Figure 24, D is a database of schema S, and A_i are applications that interface with D via S.

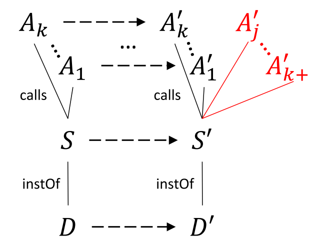


Fig. 24. Mappings in DbMM.

Figure 24 shows that a refactoring (both our notion and that of a typical Java refactoring) not only modifies schema S to S', but also updates its database D to D', and updates each existing application A_i to its semantic counterpart A'_i that references schema S'. Doing so requires a solution to the view update problem [30, 84]. Our work does not cover the refactoring of application code—this is a future problem to address. New applications $A'_{k+1} \ldots A'_j$ are subsequently added to interface with S' and its database D', possibly some applications are deleted, and the cycle of Figure 24 continues into the future.

6.8 Refactoring Text-Based DSLs

Another popular way to define an MDE metamodel, besides a [umlCD, constraints] pair, is using a purely textual DSL, which has a grammar, lexer, and parser. A DSL could be a clone of Java, where model refactoring (move method m in class C to class D) becomes much more complicated, as it must deal with methods, class member references, scoping, modularity, conditional expressions, and so on, that do not exist in umlCD metamodels.

An MDE-based refactoring engine for Java, R3, was proposed by Kim et al. [46] that parallels our work. Instead of directly refactoring an ***Abstract Syntax Tree*** (**AST**) as is usual, data on classes and their members are harvested from a program's AST and stored in a main-memory database (much like instances of ***MetaS*** store textual class and field declarations in a database). Like ***MetaS***, refactorings become tuple update operations on databases, *not* ASTs. Example: to move method m in class C to class D requires the tuple of m to update its class pointer to C (or rather a pointer to the tuple for C) to D (the tuple for D). To extract the refactored source of a program, the AST is pretty-printed, using the updated database to guide a model-to-text transformation.

Experiments showed R3 provided better refactoring extensibility, smaller memory footprint, and significantly improved performance than the Eclipse refactoring engine [46]. Whether R3's design can be used for refactoring verification remains open.

7 CONCLUSIONS

Refactoring umlCDs seemed intuitively simple to the point that correctness was "evident." ***mergeFields***$_\ominus$-***splitField***$_\ominus$ were typical: they seemed like pushovers. Sadly, this was not the case when details of a refactoring were exposed. What started as a small research endeavor kept ballooning into ever larger challenges. Just verifying a *minRef* with all of its twists-and-turns, without being overwhelmed, was initially daunting. We believe it shouldn't have to be this way.

Lessons Learned. MDE refactorings are indeed simple, but their Coq proofs are not. Significant and unexpected technical challenges surfaced with regularity as we proceeded. It is fair to say this was among the most technically challenging problems we ever faced. We offer four lessons.

First, care must be taken in defining the domain and codomain of each refactoring. Getting the correct preconditions and postconditions is crucial for verification. Example: a precondition to pull-up a field f of class B, a subclass of A, is that *all* subclasses of A contain f. If this is not the case, pull-up is an edit, not a refactoring. Far too many MDE metamodel operations (add/remove field) are labeled in IDEs as refactorings; they are embeddings or edits in our view.

Second, our choice of theorem prover was not ideal. Coq is a magnificent tool. It was appealing because it was recent, popular, and well-maintained. Further, modeling relational databases in Coq was not difficult, but it felt like an unnecessary reinvention. Coq is based on a mathematical logic that made proofs more involved than required. The chief complexities stem from properties (or constraints) defined over a structure: (1) properties are not treated as computational expressions but rather as types; and as a result (2) equivalence between two instances a and b (of the same structure) does not immediately mean that `P(a) = P(b)` for some property `P`.

Third, choosing a tool with reflection (i.e., meta-) capabilities would make the connection between metamodel and model refactorings elegant—a feature Coq lacks.

Fourth, our difficulties with Coq primarily stemmed from its low-level abstractions. We conjectured that a theorem prover of Relational Algebra ($\mathbb{R}_{\mathbb{A}}$) equalities would have simplified our task, and indeed would have helped us tap into existing database research results that are (in our opinion) far ahead of current MDE model management thinking. The reason: $\mathbb{R}_{\mathbb{A}}$ provides a unified way to express co-transformations on database schemas and their instances (read: umlCDs and their object diagram instances). Although there is now direct evidence that OCL implements a subset of $\mathbb{R}_{\mathbb{A}}$ [10], the essential operations of join and projection are missing.

Summary. Verifying the refactoring of MDE metamodels and their models has been a longstanding challenge. Prior work was hindered by choosing different correctness criteria for refactorings. Some chose embeddings [35, 36], where a refactoring $\boldsymbol{R}{:}\mathcal{M}\rightarrow\mathcal{N}$ embeds metamodel $\mathcal{M}$ into another metamodel $\mathcal{N}$ (i.e., $\mathcal{M}\hookrightarrow\mathcal{N}$) often with the help of VEs. We argued that the inverse of a refactoring is itself a refactoring, where a mutual embedding $\mathcal{M}\hookrightarrow\mathcal{N}$ and $\mathcal{N}\hookrightarrow\mathcal{M}$ leads to an equivalence—a central fact that we exploited in this article and our proofs. Prior work (mostly in database research) used a definition that two databases are *behaviorally equivalent* if they produce the same results for the same set of queries, e.g., [94]. The $\mathbb{C}_{\mathbb{T}}$ definition of equivalence that we used is more restrictive: data equivalence is required for all possible sets of queries.

Our approach to verification is incremental. We first considered minimal refactorings (a small example of a refactoring that captures its essence) *without* OCL and minimal (cardinalities, uniqueness) constraints. We then generalized our approach to consider minimal refactorings *with* minimal constraints, and again *without* OCL constraints. We discussed in Future Work parametric generalizations and context generalizations of minimal refactorings and how they could be accomplished. The refactoring of OCL constraints is still open, although prior work exists [7, 24, 39, 40, 56, 94].

Central to all these results is the framework of $\mathbb{C}_{\mathbb{T}}$ that has guided our research in a structured and incremental way; without it we could not have tackled this problem and its scope.

APPENDICES

A POLYMORPHISM AND DISTRIBUTIVITY OF METAMODEL REFACTORINGS

From the Introduction, a metamodel refactoring $\boldsymbol{R}$ transforms metamodel $\mathbb{m} = [cd, k]$ into an equivalent metamodel $\mathbb{m}' = [cd', k']$. A distributivity law—a refactoring distributes over its metamodel's components—relates $\mathbb{m}$ and $\mathbb{m}'$:

$$\boldsymbol{R}(\mathbb{m}) \;=\; \boldsymbol{R}([cd,k]) \;=\; [\boldsymbol{R}(cd),\boldsymbol{R}(k)] \;=\; [cd',k'] \;=\; \mathbb{m}'. \tag{29}$$

Three distinct interpretations of $\boldsymbol{R}$ exist in Equation (29). Let

— Θ be the domain of all MDE metamodels used in this article,
— $\mathcal{C}$ be the domain of all umlCDs, and
— $\mathcal{K}$ is the powerset of all OCL constraints, as an instance of $\mathcal{K}$ is a set of OCL constraints.

Further, there exists

— $\boldsymbol{R}{:}\Theta\to\Theta$ a general refactoring of metamodels in this article;
— $\boldsymbol{R}{:}\mathcal{C}\to\mathcal{C}$ a general refactoring of umlCDs; and
— $\boldsymbol{R}{:}\mathcal{K}\to\mathcal{K}$ a general refactoring of constraints.

Three kinds of polymorphism are recognized [98]:

(1) *Parametric Polymorphism* where methods can be written in a type-independent manner.
(2) *Subtype Polymorphism* where different classes of an inheritance hierarchy can have the same method name, each with distinct method bodies.
(3) *Ad hoc Polymorphism* a common method name given to different types.

Metamodel refactorings are examples of ad hoc polymorphism, but realize that any $\mathbb{C}_{\mathbb{T}}$ functor that maps a product of types [70] yields functions that are ad hoc polymorphic.

Example. Domain Θ is formed by the cross product of domains $\mathcal{C}$ and $\mathcal{K}$ in Figure 25(a) below; projection arrows π_C and π_K extract cd and k, respectively, from a $[cd, k]$ tuple (a.k.a. an Θ tuple) [70]. The cross product $\Theta = (\mathcal{C} \times \mathcal{K})$ forms category $\mathcal{R}$. $\boldsymbol{R}{:}\mathcal{R}\to\mathcal{R}$ is a functor from $\mathcal{R}$ to $\mathcal{R}$ (Figure 25(b)). Figure 25(c) expands $\mathcal{R}$ into its external diagram of Figure 25(a) and shows that the name $\boldsymbol{R}$ is given to all arrows in functor $\boldsymbol{R}$, exactly as in the bullet-list above.

This is the origin of the "distributivity law" used in Section 1 and Equation (29): $\boldsymbol{R}([cd, k])$ translates to $[\boldsymbol{R}(cd), \boldsymbol{R}(k)]{=}[cd', k']$; each $\boldsymbol{R}$ corresponds to a different $\boldsymbol{R}$ in Figure 25(c). Further, $\boldsymbol{R}^{-1}$ translates $[cd', k']$ in the opposite direction, where $\boldsymbol{R}^{-1}$ also has three distinct meanings. **Note:** As our work resides in a MOF universe, $\boldsymbol{R}{:}\Theta\to\Theta = I_\Theta$ is an identity function—the metamodel of all MDE metamodels is unchanged.

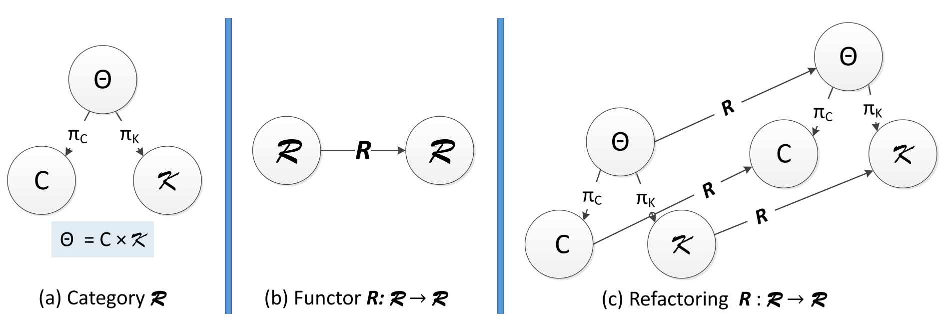


Fig. 25. Functor $\boldsymbol{R} : \mathcal{R} \to \mathcal{R}$.

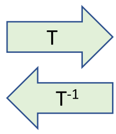
B GAMEBOARD ISOMORPHISMS

Two isomorphic and *unequal* tables are shown in Figure 26. The refactoring T and its inverse, T^{-1}, define an *abnormal* isomorphism. An abnormal refactoring is when values are translated incorrectly, but consistently.

With few exceptions, a general property that all refactorings in the literature share is the *GameBoard constraint*. On a game board, pieces can be moved to different positions on a board by stated

Dog

Name	Age	Alive
Chief	9	False
Belle	3	True



Dog'

Name	Age	Alive
DX	-9	True
DY	-3	False

Value Translations

Dog Value	Chief	Belle	9	3	True	False
Dog' Value	DX	DY	-9	-3	False	True

Fig. 26. Isomorphic tables.

rules, but the value of the piece never changes.[5] The pieces of a schema/database refactoring are user-supplied data values (*not pointers*) in its tuples. We assume every data value can be moved to a different field or tuple by rules, but a data value is never altered.

Enforcing the GameBoard rule is a metalevel constraint on proofs, which we observe: ***"data values are moved, never altered."***

C CYCLIC DATABASES

Figure 27 is a cyclic database: it is impossible to define table D without a circular definition, and Coq forbids records with circular definitions. Further, our use of embedding an entire record into a foreign key field of a Coq record doesn't work with cyclic databases, as record embedding would nest deeply in a cycle.

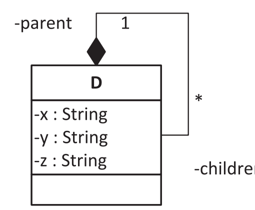


Fig. 27. A cyclic schema.

A simple approach eliminates these (and other possible) problems by making tuple-identifiers (a.k.a. keys) as explicit strings, as is common in databases [30, 84]. Consider this noncyclic Coq definition of D:

```
    Record D := mkD {  (* D tuple constructor *)
      x : string;
      y : string;
      z : string;
      parent : string; (* string identifier of parent tuple *)
      id := (x ++ y);  (* string tuple identifier *)
};.
```

A D record would be a 5-tuple, where fields $x, y, z, parent$ and id are *string*s. The last field, *id*, is the tuple identifier, which is formed by a concatenation of its x and y fields, i.e., D's primary key. (This computation tells us how to manufacture an *id* for every tuple to be inserted). Given this, one can write functions to compute association traversals (given a D record return its *parent*, or return the set of its *children*).

This is a more complicated encoding of a schema and database; we did not find a need to use it, but it was available if needed.

D PROOF OF *MERGEFIELDS*$_\ominus$-*SPLITFIELD*$_\ominus$ DATABASE REFACTORINGS

D.1 Declare Schemas

Schema $\mathbb{s}$ of Figure 1(a) has only one table, `Person`. Their Coq definitions are

```
    Record Person := mkPerson {  (* Person tuple constructor *)
     fname : string;
     lname : string;
     v_flname := (fname, lname); (* virtual element *)
    }.
```

[5]Other than promotion (pawn to queen) and capture (removal), chess follows the GameBoard constraint.

```
Record 𝕤 := mk𝕤 {              (* 𝕤 database constructor *)
 pl : list Person;
}.
```

Listing 5. Schema 𝕤 Declaration.

A VE, v_flname, is added to the table definition of Person in Listing 5 to compute a pair that encodes a full name. Symbol (:=) denotes a computation as opposed to (:) denoting a type. The other schema 𝕤' and its table, Figure 1(c), are

```
Record Person' := mkPerson' {  (* Person' tuple constructor *)
 flname   : string*string;
 v_fname := fst flname;        (* virtual element *)
 v_lname := snd flname;        (* virtual element *)
}.

Record 𝕤' := mk𝕤' {             (* 𝕤' database constructor *)
 pl' : list Person';
}.
```

As before, v_fname and v_lname are VEs. Functions fst and snd are built in and return the first and second elements of a pair, respectively.

D.2 Define Database $_{min}Ref_{\mathbb{S}}$

Function $\boldsymbol{mergeFields}_{\ominus}$:𝕤→𝕤' transforms each database instance of 𝕤 to a corresponding 𝕤' instance and $\boldsymbol{splitField}_{\ominus}$:𝕤'→𝕤 is its inverse. We use helper functions: one to translate a Person tuple to a Person′ tuple, and another to do the inverse:

```
Definition toPerson'(p: Person)   : Person' := mkPerson' (v_flname p).

Definition toPerson (p': Person') : Person  := mkPerson (v_fname p') (v_lname p').
```

toPerson′ constructs a new Person' by using the virtual field v_flname and $toPerson$ constructs a new Person using virtual fields v_fname and v_lname. Functions $\boldsymbol{mergeFields}_{\ominus}$ and $\boldsymbol{splitField}_{\ominus}$ become

```
Definition mergeFields⊖ (db : 𝕤)  : 𝕤' := mk𝕤' (map toPerson' (pl db)).

Definition splitField⊖ (db' :𝕤')  : 𝕤  := mk𝕤 (map toPerson (pl' db')).
```

where map is a Coq built-in function with two parameters, a function and a list, and applies the function to every element in that list.

D.3 Invertibility Theorems

$\boldsymbol{mergeFields}_{\ominus}$ and $\boldsymbol{splitField}_{\ominus}$ are inverses of each other by proving these round-trip theorems:

```
Theorem 𝕤_roundTrip:  forall (db : 𝕤),    db  = splitField⊖(mergeFields⊖ db).

Theorem 𝕤'_roundTrip: forall (db' : 𝕤'), db' = mergeFields⊖(splitField⊖ db').
```

Listing 6. Main Theorems for Database Refactoring.

and define two lemmas: roundTripPerson and roundTripPerson′ to show that toPerson and toPerson′ are inverses of each other.

```
Lemma roundTripPerson  : forall (p : Person), p = (toPerson (toPerson' p)).

Lemma roundTripPerson' : forall (p' : Person'), p' = (toPerson' (toPerson p')).
```

D.4 Proof Details

The proof approach for these lemmas (1) eliminate the universal quantification (`forall`) by assuming the input p using the keyword *intros*; (2) *destruct p* by exposing its internal structure using the *destruct* tactic; (3) use the *auto* tactic to replace the current subgoal with its definition; and (4) show that equality holds. Below is a proof script with the current state of the proof (shown in $\boxed{\text{box}}$) after executing each line.

```
Lemma roundTripPerson : forall (p : Person),
   p = (toPerson (toPerson' p)).
Proof.
```

```
1 subgoal
forall p : Person, p = toPerson (toPerson' p)
```

```
intros p.                (* assume the input and call it p *)
```

```
1 subgoal
p : Person
______________________________________(1/1)
p = toPerson (toPerson' p)
```

```
destruct p.                (* break p to its basic parts *)
```

```
1 subgoal
fname0, lname0 : string
______________________________________(1/1)
{| fname := fname0; lname := lname0 |} =
toPerson (toPerson' {| fname := fname0; lname := lname0 |})
```

```
auto.                           (* simplify and discharge if possible *)
```

```
No more subgoals.
```

```
Qed.
```

The other lemma, `roundTripPerson'`, is similar.

The proof of the main theorems (Listing 6) uses these lemmas but the proof is slightly different as it deals with lists and induction. The following Coq proof script shows the resulting state inside a $\boxed{\text{box}}$ after each line:

```
Theorem 𝕤_RoundTrip : forall (db : 𝕤),
 db = (splitField⊖ (mergeFields⊖ db)).
Proof.
```

```
1 subgoal
______________________________________(1/1)
forall db : 𝕤, db = splitField⊖ (mergeFields⊖ db)
```

```
destruct db as [elems]. (* destruct db to its lone field. Assume its value to be 'elems'*)
```

```
1 subgoal
elems : list Person
______________________________________(1/1)
{| pl := elems |} = splitField⊖ (mergeFields⊖ {| pl := elems |})
```

The *destruct* tactic in Line 7 eliminates the universal quantifier, and destructs *db* to its only field (i.e., *pl*). The value of *pl* is assigned to some list of *Person*s which we have named *elems*.

```
8   unfold splitField⊖, mergeFields⊖ (* unfold to their definitions *)
```

```
1 subgoal
elems : list Person
______________________________________(1/1)
{| pl := elems |} =
{| pl := map toPerson (pl' {| pl' := map toPerson' (pl {| pl := elems |}) |}) |}
```

Next, the current subgoal is updated and *elems* is assumed (i.e., it is added to the hypothesis environment). Line 8 instructs Coq to *unfold* the definitions of ***mergeFields***$_\ominus$ and ***splitField***$_\ominus$ which updates the current subgoal as shown in the box after Line 8.

```
9   f_equal. (* compares corresponding expressions *)
```

```
1 subgoal
elems : list Person
______________________________________(1/1)
elems = map toPerson (pl' {| pl' := map toPerson' (pl {| pl := elems |}) |})
```

To compare corresponding expressions and drop `pl` to the left of the assignment symbol (:=), the `f_equal` tactic is used. The result is shown in the box following Line 9.

```
10  induction elems. (* generates two cases: when elems is empty and when it is not*)
```

```
2 subgoals
______________________________________(1/2)
[] = map toPerson (pl' {| pl' := map toPerson' (pl {| pl := [] |}) |})
______________________________________(2/2)
a :: elems = map toPerson (pl' {| pl' := map toPerson' (pl {| pl := a :: elems |}) |})
```

```
16  - auto.            (* solves the first subgoal *)
17  - simpl. f_equal.  (* generates two cases: base and induction step *)
```

```
2 subgoals
a : Person
elems : list Person
IHelems : elems = map toPerson (pl' {| pl' := map toPerson' (pl {| pl := elems |}) |})
______________________________________(1/2)
a = toPerson (toPerson' a)
______________________________________(2/2)
elems = map toPerson (map toPerson' elems)
```

At this point, induction is used on *elems* in Line 10. By the definition of lists in Coq, there are two ways to create a list: (1) creating an empty list (using *nil*), and (2) adding an element to an existing list (using *cons*). Therefore, a subgoal is generated for each case. The first is straightforward and is solved using the *auto* tactic. Using *simpl* and `f_equal` in Line 17, the second subgoal is further split into two subgoals: base case and induction step (as shown in the box after Line 17).

```
18    + apply RoundTripPerson. (* solves base case *)
```

```
1 subgoal
a : Person
elems : list Person
IHelems : elems = map toPerson (pl' {| pl' := map toPerson' (pl {| pl := elems |}) |})
______________________________________(1/1)
elems = map toPerson (map toPerson' elems)
```

```
19    + assumption. (* applies 'IHelems' hypothesis *)
20    Qed.
```

The base case uses the previously defined lemma *RoundTripPerson* as it matches the (first) subgoal. The current state updates the box shown after Line 18. Now the subgoal intuitively matches the hypothesis *IHelems*[6] generated by the induction step. The *assumption* tactic in Line 19 instructs Coq to look at the list of hypotheses and try to match the subgoal with a suitable hypothesis. This ends our proof and concludes with *Qed*.

E PROOF OF *MERGEFIELDS*$_\ominus$-*SPLITFIELD*$_\ominus$ SCHEMA REFACTORINGS

E.1 Declare Schemas

The ***MetaS*** encodings of 𝕤 and 𝕤' were defined in Listing 4 of Section 3.2.

E.2 Define Schema *minRefs*

mergeFields$_\ominus$ takes schema 𝕤 and returns schema 𝕤', and ***splitField***$_\ominus$ does the inverse:

$$\boldsymbol{mergeFields}_\ominus : \{\mathbb{s}\} \longrightarrow \{\mathbb{s}'\} \qquad \bigwedge \qquad \boldsymbol{splitField}_\ominus : \{\mathbb{s}'\} \longrightarrow \{\mathbb{s}\}. \tag{30}$$

Transforming 𝕤 to 𝕤' requires replacing columns $c1$ and $c2$ with $c3$. Everything else remains unchanged. This is reflected in the code below:

```
Definition mergefields (s : MetaS) (x y z: Column) : MetaS :=
mkMetaS (tbls s) (add z (rmv y (rmv x (cols s)))).
Goal (mergefields 𝕤 c1 c2 c3) = 𝕤'.       (* applying mergeFields to 𝕤 returns 𝕤' *)
Proof.
 unfold mergefields, 𝕤'.
 f_equal.
Qed.
```

The function *mergeFields*, modifies its input schema, in our case 𝕤 which has one table, by updating its list of *Column*s: x and y (denoting $fname$ and $lname$) are removed (using the function *rmv*) and z (denoting $flname$) is added (using the function *add*). *splitField* is the inverse of *mergeFields*:

```
Definition splitField (s : MetaS) (z x y: Column) : MetaS :=
mkMetaS (tbls s) (add x (add y (rmv z (cols s)))).
Goal (splitField 𝕤' c3 c1 c2) = 𝕤. (* applying splitField to 𝕤' returns 𝕤 *)
Proof.
 unfold s1, splitField.
 f_equal.
Qed.
```

[6]The name *IHelems* is auto generated by Coq which represents the inductive hypothesis of the list *elems*.

E.3 Theorems and Proofs

The invertibility theorems of the ***mergeFields***$_\ominus$-***splitField***$_\ominus$ schema *minRefs* are straightforward and so are their proofs:

```
Theorem th1:
 𝕤 = splitfield (mergefields 𝕤 c1 c2 c3) c3 c1 c2.
Proof.
 unfold 𝕤, splitfield, mergefields.
 f_equal.
Qed.

Theorem th2:
 𝕤' = mergefields (splitfield 𝕤' c3 c1 c2) c1 c2 c3.
Proof.
 unfold 𝕤', splitfield, mergefields.
 f_equal.
Qed.
```

The first theorem says that applying ***MergeFields***$_\ominus$ to 𝕤 then ***SplitField***$_\ominus$ recovers 𝕤. Similarly, applying ***SplitField***$_\ominus$ then ***MergeFields***$_\ominus$ to 𝕤' yields 𝕤' again. The proof script for both theorems shares the same idea: definitions are unfolded and their corresponding field values are compared using *f_equal* tactic. This tactic also solves equivalent values, which concludes the proof.

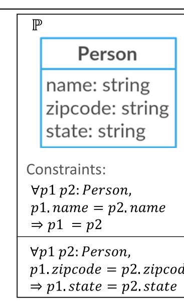
F PROOF OF *EXTRACT*$_\ominus$-*INLINE*$_\ominus$ DATABASE REFACTORINGS

F.1 Declare Schemas

The Coq specification of schema ℙ is

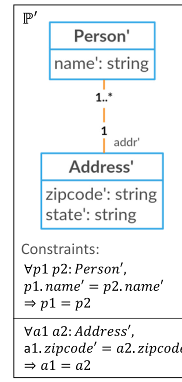
```
Record Person := mkPerson {                  (* Person tuple constr. *)
 name    : string;
 zipcode : string;
 state   : string;
}.

Record ℙ := mkℙ {                            (* ℙ database constructor *)
 pl        : list Person;
 personKey : forall p1 p2,
             In p1 pl → In p2 pl →           (* name is primary key *)
             (name p1) = (name p2) →         p1 = p2;
 sameState : forall p1 p2,
             In p1 pl → In p2 pl →           (* any 2 Persons in pl *)
             (zipcode p1) = (zipcode p2) →   (* with same zipcode value *)
             (state p1) = (state p2);        (* must share same state value *)
}.
```



And now schema $\mathbb{P}'$ with constraints personKey′, addressKey′, and card′:

```
Record Address' := mkAddress' { (* Address' tuple constructor *)
 zipcode' : string;
 state'   : string;
}.
Record Person' := mkPerson' {   (* Person' tuple constructor *)
 name'    : string;
 addr'    : Address';           (* each Person has one Address' *)
}.
Record ℙ' := mkℙ' {             (* ℙ' database constructor *)
 pl'         : list Person';
 al'         : list Address';
 personKey'  : forall p1 p2,
               In p1 pl' → In p2 pl' →            (* name' is primary key *)
               (name' p1) = (name' p2) → p1 = p2;
 addressKey' : forall a1 a2,
               In a1 al' → In a2 al' →             (* zipcode' is primary key *)
               (zipcode' a1) = (zipcode' a2) →   a1 = a2;
 card'       : al' = nodup addr_dec (map addr' pl'); (* at-least-one cardinality *)
}.
```



Listing 7. Modeling the database of schema $\mathbb{P}'$ in Coq.

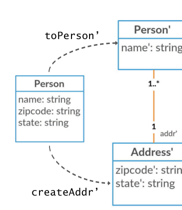
Note: The card′ constraint says the result of collecting all Address′ tuples from Person′ tuples and removing duplicates (via the Coq built-in nodup) yields the Address′ list al′. In other words, each Address′ tuple is referenced.

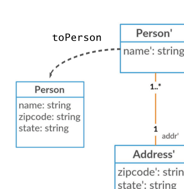
F.2 Define Database *minRefs*

Given an instance of Person, corresponding instances of Person′ and Address′ can be derived. Line 2 below, createAddr′, translates a Person to an Address′ by extracting the zipcode and state values from Person p. On Line 4, a Person′ instance is created by extracting the person's name (name p) and its Address′ tuple (createAddr′ p).

Conversely, a Person′ instance p' has an embedded Address′ instance and can be translated to a Person instance. Line 7, toPerson, creates a Person by *pattern matching*, which breaks p' into its components: the matching constructor mkPerson′, a name n, and an Address′ which is further decomposed into its constructor mkAddress′, a zipcode z, and state c. These values are then used to create a Person instance. **Note**: ⇒ is used in pattern matching branches. It is different from → which is used to denote logical implication or to separate function types.

```
1   (* From Person to Address' -- Left to Right *)
2   Definition createAddr' (p:Person) := mkAddress' (zipcode p) (state p).
3   (* From Person to Person' *)
4   Definition toPerson' (p:Person) : Person':=
5    mkPerson' (name p) (createAddr' p).
6   (* From Person' to Person -- Right to Left*)
7   Definition toPerson (p':Person'): Person:=
8    match p' with
9     | mkPerson' n (mkAddress' z c) ⇒ mkPerson n z c
10   end.
```





Transformation to$\mathbb{P}'$ converts a $\mathbb{P}$ database to a $\mathbb{P}'$ database. Converting a list of `Person` tuples to a list of `Person′` tuples and then to a list of `Address′` tuples is easy. Showing that constraints `sameState`, `addressKey′`, and `card′` also hold is another matter. We show the important steps to prove `personKey′` below; proofs of other constraints follow a similar pattern. Given the list translation

$$pl' = (map\ toPerson'\ pl), \tag{31}$$

constraint `personKey′` becomes after substituting (31)

```
forall p1 p2 : person',
In p1 (map toPerson' pl) →
In p2 (map toPerson' pl) →
name' p1 = name' p2 →
p1 = p2.
```

A proof follows by expanding `toPerson′` and relying on constraints from $\mathbb{P}$, here: `personKey`.

F.3 Proof Details

The transformation definitions between the $\mathbb{P}$ and $\mathbb{P}'$ are

```
Definition toℙ(db':ℙ′) : ℙ  := mkℙ(db').
Definition toℙ′(db:ℙ)  : ℙ′ := mkℙ′(db).
```

We first define helper lemmas to prove invertibility between table transformations: `toPerson`, `toPerson′`, and `createAddr′`:

```
Lemma reconstructPerson  : forall p:Person, p = toPerson(toPerson' p).
Lemma reconstructPerson' : forall p':Person', p' = toPerson'(toPerson p').
Lemma reconstructAddr'   : forall p':Person', addr' p' = createAddr'(toPerson p').
```

The proof scripts of these lemmas are straightforward. However, because the database transformations `toℙ` and `toℙ′` involve constraints, their invertibility theorems `reconstructℙ` and `reconstructℙ′` require special treatment. We now explain the steps needed to prove `reconstructℙ`. The same approach is used by `reconstructℙ′` and is omitted. The theorems are as follows:

```
Theorem reconstructℙ  : forall db:ℙ, db = toℙ(toℙ' db).
Theorem reconstructℙ' : forall db':ℙ′, db' = toℙ'(toℙ db').
```

Recall that a constraint must be transported to a suitable type before equality can be established (Section 4.1). In our case, two $\mathbb{P}$ instances, *db1* and *db2*, that encode the *same* data will have mismatched `personKey` types and mismatched `sameState` types. With these transports, we can prove the equivalence of *db1* and *db2*:

```
(* Generalize the definitions of 'personKey' and 'sameState' constraints for readability *)
Definition personKeyDef (pl: list person):=
  forall p1 p2,  In p1 pl → In p2 pl → name p1 = name p2 → p1 = p2.

Definition sameStateDef (pl: list Person):=
 forall p1 p2,  In p1 pl → In p2 pl → (zipcode p1) = (zipcode p2) → (state p1) = ( state p2).

(* Transport 'personKey' *)
Definition transportPersonKey {p1 p2: list person}:
 p1 = p2 →                (* if the two lists are equivalent*)
 personKeyDef p1 →        (* and if (personKey p1) holds *)
 personKeyDef p2.         (* then (personKey p2) must hold *)
Proof.
 intros H1 H2.
 rewrite H1 in H2.
 assumption.
```

```
Defined.
(* Transport 'sameState' is almost identical to above *)
```

Our main theorem *reconstruct*ℙ requires us to prove `db` and `(toℙerson (toℙerson′ db))` are equivalent. The *eqDB* lemma (Line 1 below) is used for this purpose and requires four inputs:

(1) two ℙ instances—in our case, these are `db` and `(toℙ(toℙ′ db));`
(2) a proof that their `pl` lists are equivalent—i.e., a proof that `(pl db = pl (toℙ (toℙ′ db)));`
(3) a proof that their `personKey` proofs are equivalent—solved using the proof irrelevance axiom (Section 4.1); and
(4) a proof that their `sameState` proofs are equivalent.

```
1  Lemma eqDB (db1 db2 : ℙ)       (* given two ℙ instances*)
2   (p: pl db1 = pl db2)          (* with equal Person lists*)
3                                 (* and with same constraints after transportation *)
4   (q: transportPersonKey p (personKey db1) = personKey db2)
5   (r: transportSameState p (sameState db1) = sameState db2):
6   db1 = db2.                    (* then these two instances are equal *)
7  Proof.
8   destruct db1; destruct db2.
9   simpl in * |- *.
10  destruct p, q, r.
11  reflexivity.
12 Qed.
```

The proof of the second point is discharged using induction and the *reconstruct*ℙ lemma:

```
Lemma eqPersonList (db: ℙ): pl db = pl (toℙ (toℙ' db)).
Proof.
 unfold toℙ', toℙ.
 destruct db as [PL PP PF]. (* assign PL, PP, and PF to pl, personKey, and sameState *)
 simpl.
 induction PL as [| p pl].
 - reflexivity.
 - simpl. f_equal.
 + apply reconstructPerson.
 + firstorder.
Qed.
```

Now, the proof of the main theorem is

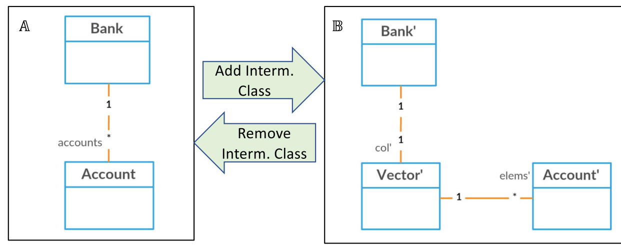
```
Theorem reconstructℙ: forall db : ℙ, db = toℙ (toℙ' db).
Proof.
 intros db.                       (* 'pl' field is equal in both instances *)
 apply (eqDB db (toℙ (toℙ' db)) (eqPersonList db)).
 apply proof_irrelevance.         (* 'personKey' holds for both instances *)
 apply proof_irrelevance.         (* 'sameState' holds for both instances *)
Qed.
```

The theorem says: for any ℙ instance, *db*, it is always equal to its reconstructed version after round-tripping, `toℙ(toℙ′ db)`. The proof of the inverse transformation, `toℙ′`, is not much different. The same approach is used by `reconstructℙ′` and is omitted.

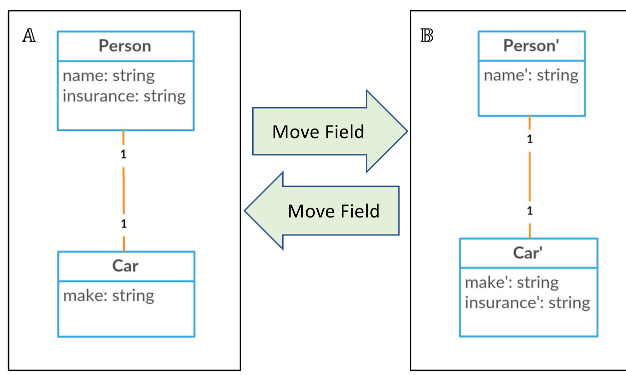
G REFACTORINGS THAT HAVE BEEN VERIFIED

Beyond the refactorings covered in the body of this article, we have proofs for the following:

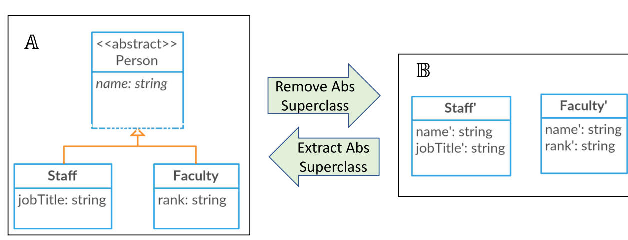
(1) Add/Remove Intermediate Class. This refactoring was taken from [35] as it could only be proven by its authors as a one-way embedding. We used this example to prove bi-directional embeddings.



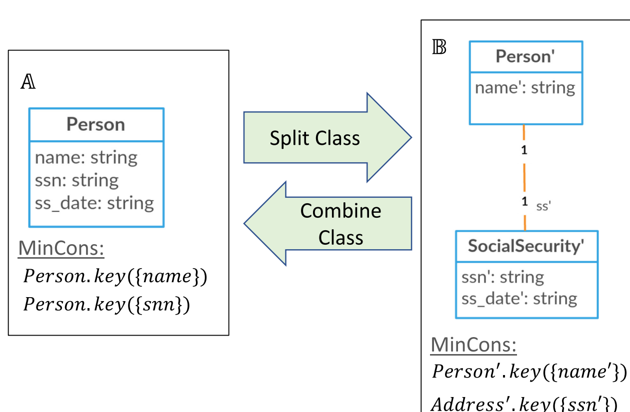
(2) Move Field. Moves a field from one class to another via an existing 1:1 association. A field F cannot be moved if the target class already has a field named F.



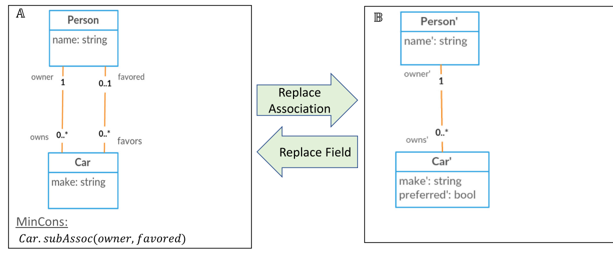
(3) Remove/Extract Abstract Superclass. *Person* is an abstract class without associations and is an immediate subclass of *Object*. This refactoring pushes the contents of *Person* down into its subclasses.



(4) Split/Combine Class. A class has two key fields *name* and *ssn* with functional dependencies $ssn \rightarrow \{name, ss\_date\}$ and $name \rightarrow \{ssn, ss\_date\}$. This refactoring splits the *Person* class into two classes connected by a 1:1 association.

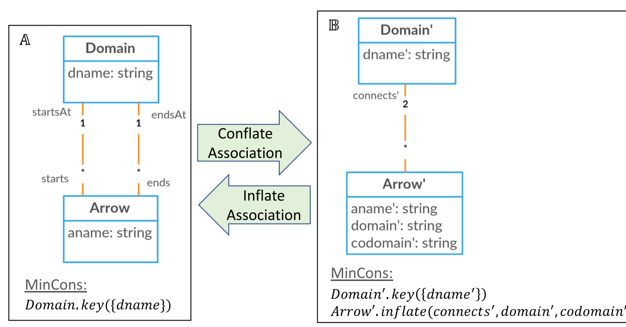


(5) Replace Sub-Association with Field. An association can be a sub-association of another. In $\mathbb{A}$, a *Person* owns many *Car*s, but a subset of these *Car*s can be `favored`. This refactoring replaces the sub-association with a `preferred` Boolean attribute of *Car′* that indicates if the car is favored.



Note: The *minCons* in $\mathbb{A}$ declares the `(favored--favors)` association is a subassociation of `(owner--owns)`.

(6) Conflate/Inflate Association. This is an odd refactoring submitted by students in Batory's undergrad course, and was used as a stress test for our approach. $\mathbb{A}$ defines a category diagram, where nodes have unique `domain` names and edges (which start at one node and end at the same or another node) are arrows. Two associations are used in $\mathbb{A}$ and are squashed into a single 2:* association in $\mathbb{B}$ by adding fields `domain` and `codomain` to encode arrow direction information.



Note: The *minCons* in $\mathbb{B}$ is an abbreviation of

$$\forall a' \in Arrow' : fst(a'.connects').dname' = a'.domain' \wedge snd(a'.connects').dname' = a'.codomain.$$

ACKNOWLEDGMENTS

We thank Jongwook Kim and the referees for their help with our article.

REFERENCES

[1] F. Allilaire and F. Jouault. 2007. Families to Persons: A Simple Illustration of Model-to-Model Transformation. https://www.eclipse.org/atl/documentation/old/ATLUseCase_Families2Persons.pdf.
[2] N. Altoyan. 2020. *MDE Refactorings: A Categorical Framework with Proofs, Tools, and Implementations*. Ph.D. Dissertation. University of Texas at Austin.
[3] N. Altoyan. 2022. Coq Proof Scripts for Metamodel Co-Refactings. https://zenodo.org/record/6645716#.YqoHOnbMKUl.
[4] K. Anastasakis, B. Bordbar, and J. Küster. 2007. Analysis of model transformations via alloy. In *MoDeVVa*.
[5] S. Andova, M. van den Brand, L. Engelen, and T. Verhoeff. 2012. MDE basics with a DSL focus. In *Formal Methods for Model-Driven Engineering*, M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.), Lecture Notes in Computer Science, Vol. 7320. Springer-Verlag.
[6] P. Atzeni, G. Ausiello, C. Batini, and M. Moscarini. 1982. Inclusion and equivalence between relational database schemata. *Theoretical Computer Science* 19 (Sept. 1982), 267–285.
[7] T. Baar and S. Marković. 2007. A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In *PSI*.
[8] D. Batory. 2020. *Automated Software Design Volume 1* (2nd ed.). Lulu.com.
[9] D. Batory. 2021. Conversation About Theorem Proving with Warren Hunt. Private communication.
[10] D. Batory and N. Altoyan. 2020. Aocl: A pure-java constraint and transformation langauge for MDE. In *MODELSWARD*.

[11] D. Batory and M. Azanza. 2017. Teaching model-driven engineering from a relational database perspective. *Software and Systems Modeling* 16 (May 2017), 443–467.
[12] D. Batory and E. Börger. 2008. Modularizing theorems for software product lines: The JBook case study. *Journal of Universal Computer Science* 14, 12 (June 2008), 2059–2082.
[13] D. Batory, E. Latimer, and M. Azanza. 2013. Teaching model driven engineering from a relational database perspective. In *MODELS*.
[14] K. Beck and C. Andres. 2004. *Extreme Programming Explained: Embrace Change* (2nd ed.). Addison-Wesley.
[15] V. Benzaken, É. Contejean, and S. Dumbrava. 2014. A Coq formalization of the relational data model. In *ESOP*.
[16] H. Berg and I. Yuh. 2017. Generic Metamodel Refactoring with Automatic Detection of Applicability and Co-Evolution of Artefacts. https://www.duo.uio.no/handle/10852/54485?locale-attribute=no.
[17] R. Berghammer and G. Schmidt. 1993. *Relational Specifications*. Technical Report. Banach Center Publications, Vol. 28, Institute of Mathematics, Polish Academy of Sciences.
[18] P. Bernstein, T. Green, S. Melnik, and A. Nash. 2008. Implementing mapping composition. *VLDB Journal* (June 2008).
[19] P. Bernstein and S. Melnik. 2007. Model management 2.0: Manipulating richer mappings. In *ACM SIGMOD*.
[20] K. Berramla, E. Deba, and M. Senouci. 2015. Formal validation of model transformation with Coq proof assistant. In *NTIC*.
[21] Y. Bertot and P. Castéran. 2013. *Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions*. Springer Science & Business Media.
[22] J. Bézivin. 2006. Model driven engineering: An emerging technical space. In *GTTSE*, R. Lämmel, J. Saraiva, and J. Visser (Eds.). Springer, Berlin.
[23] M. Brambilla, J. Cabot, M. Wimmer, and L. Baresi. 2017. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan and Claypool.
[24] J. Cabot and E. Teniente. 2007. Transformation techniques for OCL constraints. *Science of Computer Programming* 68 (2007), 179–195.
[25] D. Calegari, C. Lunas, N. Szasz, and Á. Tasistro. 2011. A type-theoretic framework for certified model transformations. In *SBMF*.
[26] V. Costa, R. Monteiro, and L. Murtao. 2014. Detecting semantic equivalence in UML class diagrams. In *SEKE*.
[27] E. de Kogel. 1993. *Relational Algebra and Equational Proofs*. Technical Report 93/23. Eindhoven University of Technology.
[28] B. Delaware, W. Cook, and D. Batory. 2011. Theorem proving for product lines. In *OOPSLA/SPLASH*.
[29] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Täntzer. 2007. Information preserving bidirectional model transformations. In *FASE*.
[30] R. Elmasri and S. Navathe. 2010. *Fundamentals of Database Systems*. Addison-Wesley.
[31] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 2000. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
[32] M. Fowler and S. Kendall. 1997. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley Longman Ltd.
[33] R. France, S. Ghosh, E. Song, and D. Kim. 2003. A metamodeling approach to pattern-based model refactoring. *IEEE Software* (Sept. 2003).
[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
[35] R. Gheyi, T. Massoni, and P. Borba. 2005. A rigorous approach for proving model refactorings. In *ASE*.
[36] R. Gheyi, T. Massoni, and P. Borba. 2005. An abstract equivalence notion for object models. *Electronic Notes in Theoretical Computer Science* (May 2005).
[37] M. Golobisky and A. Vecchietti. 2005. Mapping UML class diagrams into object-relational schemas. In *ASSE*.
[38] W. Griswold. 1991. *Program Restructuring as An Aid to Software Maintenance*. Ph.D. Dissertation. University of Washington.
[39] K. Hassam, S. Sadou, and R. Fleurquin. 2010. Adapting OCL constraints after a refactoring of their model using an MDE process. In *BENEVOL*.
[40] F. Hermann, H. Ehrig, U. Golas, and F. Orejaso. 2012. Formal analysis of model transformations based on triple graph grammars. *Mathematical Structures in Computer Science* 24 (Jan. 2012).
[41] M. Herrmannsdörfer, S. Benz, and E. Jürgens. 2009. COPE—automating coupled evolution of metamodels and models. In *ECOOP*.
[42] D. Jackson. 2002. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology* 11 (April 2002), 256–290.
[43] J. Kerievsky. 2004. *Refactoring to Patterns*. Addison-Wesley.
[44] B. Kernighan and D. Ritchie. 1989. *The C Programming Language*. Prentice Hall.

[45] J. Kim, D. Batory, and D. Dig. 2015. Scripting parametric refactorings in Java to retrofit design patterns. In *ICSME*.
[46] J. Kim, D. Batory, D. Dig, and M. Azanza. 2016. Improving refactoring speed by 10X. In *ICSE*.
[47] J. Kim, D. Batory, D. Dig, and M. Azanza. 2019. Code transformation issues in move-instance-method refactorings. In *IWoR*.
[48] H. König, M. Löwe, and C. Schulz. 2011. Model transformation and induced instance migration: A universal framework. In *SBMF*.
[49] E. Lacker, J. Kim, A. Kumar, L. Chandrashekar, S. Paramaiahgari, and J. Howard. 2021. Statistical analysis of refactoring bug reports in eclipse bugzilla. In *IWoR*.
[50] J. De Lara, E. Guerra, and J. Cuadrado. 2014. When and how to use multilevel modelling. *ACM ACM Transactions on Software Engineering and Methodology* 24 (Dec. 2014), 1–46.
[51] H. Ledang and H. Dubois. 2010. Proving model transformations. In *TASE*.
[52] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb. 2015. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. *Software Quality Journal* 25 (Aug. 2015).
[53] F. Mantz, S. Jurack, and G. Täntzer. 2012. Graph transformation concepts for meta-model evolution guaranteeing permanent type conformance throughout model migration. In *AGTIVE*.
[54] S. Maoz, J. Ringert, and B. Rumpe. 2011. CD2Alloy: Class diagrams analysis using alloy revisited. In *MODELS*.
[55] S. Maoz, J. Ringert, and B. Rumpe. 2011. CDDiff: Semantic differencing for class diagrams. In *ECOOP*.
[56] S. Marković and T. Baar. 2008. Refactoring OCL annotated UML class diagrams. *Software and Systems Modeling* 7 (Feb. 2008), 25–47.
[57] R. Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.
[58] T. Massoni, R. Gheyi, and P. Borba. 2005. Formal refactoring for UML class diagrams. In *SBES*.
[59] S. Melnik. 1998. *Generic Model Management*, Lecture Notes in Computer Science, Vol. 2967. Springer-Verlag.
[60] S. Melnik. 2004. *Generic Model Management: Concepts and Algorithms*. Lecture Notes in Computer Science. Springer-Verlag.
[61] S. Melnik, A. Adya, and P. Bernstein. 2008. Compiling mappings to bridge applications and databases. *ACM Transactions on Database Systems* 33 (Nov. 2008), 1–50.
[62] Microsoft. 2020. MS Access. https://en.wikipedia.org/wiki/Microsoft_Access.
[63] M. Mohammed and A. Mohammad. 2015. UML model refactoring: A systematic literature review. *Empirical Software Engineering* (Jan. 2015).
[64] W. Mok and D. Paper. 2001. On transformations from UML models to object-relational databases. In *HICSS*.
[65] T. Neward. 2006. The Vietnam of Computer Science. https://blogs.tedneward.com/post/the-vietnam-of-computer-science/.
[66] Niols. 2016. 'f_equal' Isn't Doing Anything. (2016). Retrieved November 21, 2019 from https://cstheory.stackexchange.com/questions/33743/f-equal-isnt-doing-anything.
[67] O. (OMG). 2006. Meta-Object Facility (MOF) Specification, Version 2.0. OMG Document Number formal/2006-01-01 http://www.omg.org/spec/MOF/2.0.
[68] B. Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
[69] R. Paige and J. Ostroff. 2001. Metamodelling and conformance checking with PVS. In *FASE*.
[70] B. Pierce. 1991. *Basic Category Theory for Computer Scientists*. MIT Press.
[71] J. Pilgrim, B. Ulke, A. Thies, and F. Steimann. 2013. Model/code co-refactoring: An MDE approach. In *ASE*.
[72] R. Pottinger and P. Bernstein. 2008. Schema merging and mapping creation for relational sources. In *EDBT*.
[73] R. Pottinger and P. Bernstein. 2009. Associativity and commutativity in generic merge. In *Mylopoulos Festschrift, Conceptual Modeling: Foundations and Applications*, A. Borgida et al. (Eds.), Lecture Notes in Computer Science, Vol. 5600. Springer-Verlag.
[74] A. Queralt and E. Teniente. 2006. Reasoning on UML class diagrams with OCL constraints. In *ER*.
[75] D. Roberts. 1999. *Practical Analysis for Refactoring*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
[76] L. Rose et al. 2010. A comparison of model migration tools. In *MODELS*.
[77] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. 2010. Model migration with epsilon flock. In *ICMT*.
[78] K. Rubin. 2012. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley.
[79] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. 2012. A formal approach to the specification and transformation of constraints in MDE. *The Journal of Logic and Algebraic Programming* (March 2012).
[80] D. Sannella and M. Wirsing. 1983. A kernel language for algebraic specification and implementation—extended abstract. In *Foundations of Computation Theory*. Springer, Berlin.
[81] M. Schäfer, A. Thies, F. Steimann, and F. Tip. 2012. A comprehensive approach to naming and accessibility in refactoring Java programs. *IEEE Transactions on Software Engineering* (Nov. 2012).

[82] C. Schulz, M. Löwe, and H. König. 2011. A categorical framework for the transformation of object-oriented systems: Models and data. *Journal of Symbolic Computation* (March 2011).
[83] B. Selic. 2012. The less well known UML. In *SFM*, M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.).
[84] A. Silberschatz, H. Korth, and S. Sudarshan. 2006. *Database System Concepts* (5th ed.). McGraw-Hill, Inc.
[85] I. Sommerville. 1995. *Software Engineering* (5th ed.). Addison Wesley Longman Publishing Co., Inc., Chapter: Algebraic Specification.
[86] R. Stärk, J. Schmid, and E. Börger. 2001. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag.
[87] D. Steinberg and F. Budinsky. 2008. *EMF: Eclipse Modeling Framework* (2nd ed.). Addison-Wesley Professional.
[88] R. Sträten, V. Jonckers, and T. Mens. 2007. A formal approach to model refactoring and model refinement. In *SoSYM*.
[89] N. Sultana and S. Thompson. 2008. Mechanical verification of refactorings. In *PEPM*.
[90] R. Sunderraman. 2007. *Oracle 10g Programming: A Primer.* Addison-Wesley Longman Publishing Co., Inc.
[91] G. Täntzer, F. Mantz, and Y. Lamo. 2012. Co-transformation of graphs and type graphs with application to model co-evolution. In *ICGT*.
[92] L. Tokuda and D. Batory. 1999. Evolving object-oriented designs with refactorings. In *ASE*.
[93] G. Wachsmuth. 2007. Metamodel adaptation and model co-adaptation. In *ECOOP*.
[94] Y. Wang, I. Dillig, S. Lahiri, and W. Cook. 2018. Verifying equivalence of database-driven applications. In *POPL*.
[95] Wikipedia. 2018. Partial Function. https://en.wikipedia.org/wiki/Partial_function.
[96] Wikipedia. 2019. Classical Logic. Retrieved December 12, 2019 from https://en.wikipedia.org/wiki/Classical_logic.
[97] Wikipedia. 2019. Intuitionistic Logic. (2019). Retrieved September 11, 2019 from https://en.wikipedia.org/wiki/Intuitionistic_logic.
[98] Wikipedia. 2020. Polymorphism. https://en.wikipedia.org/wiki/Polymorphism_(computer_science).
[99] Wikipedia. 2021. Domain-Specific Language. https://en.wikipedia.org/wiki/Domain-specific_language.
[100] I. Yu and H. Berg. 2015. A framework for metamodel composition and adaptation with conformance-preserving model migration. In *MODELSWARD*.

Received 10 February 2021; revised 25 May 2022; accepted 11 June 2022