Constraints in Object-Oriented Databases

Joachim Biskup Torsten Polle

Universität Dortmund, FB Informatik, D-44221 Dortmund

{biskup,polle}@ls6.informatik.uni-dortmund.de

6th September 2000

Abstract

Normal forms in relational database theory, like 3NF or BCNF, are defined by means of semantic contraints. Since for these constraints sound and complete axiomatisations exist and, additionally, for some of these constraints the implication problem is decidable, computer aided database design is possible for relational data models. Object-oriented database theory lacks such normal forms, partly because neither a classification of semantic constraints nor sound and complete axiomatisations exist.

In this work we present three classes of semantic constraints for object-oriented data models and show that these constraints have a sound and complete axiomatisation. Thus we prepare the grounds for normal forms in object-oriented data models and subsequently for computer aided object-oriented database design.

1 Introduction

The theory of database design for relational data models identifies a number of properties to characterise good database schemas. These properties lead then to normal forms for database schemas to exclude undesirable properties. The design theory for object-oriented data models lacks such clear and well-understood properties, or to be more precise their formalisation and their syntactic characterisation. Semantic constraints especially functional dependencies play in the relational data model a vital rôle in the syntactic characterisation of a number of properties [Bis95]. Biskup et. al. [BMPS96] show that functional dependencies can be deployed to guide a decomposition called *pivoting* for object-oriented schemas, but the decomposition does not produce equivalent schemas unless the functional dependencies are in a special form. We can lift these restrictions if we use *path functional dependencies* [Wed89, Wed92], an extension of functional dependencies for object-oriented data models [Pol00, BP00a, BP00b]. In order to work with path functional dependencies we need inference rules, which exist for path functional dependencies alone [Wed89, Wed92]. Unfortunately, to produce equivalent

schemas, pivoting introduces a second kind of semantic constraints, *onto constraints*, whose nature is more like inclusion dependencies as known in the relational data model. But then onto constraints have an impact on the logical implication of path functional dependencies; a fact, which is not captured by the inference rules proposed by Weddell. In this work, we close this gap by introducing inference rules working for a combination of *class inclusion constraints*, *onto constraints* and *path functional dependencies.* Class inclusion constraints come into play due to the interplay between signature declarations for classes in a schema and onto constraints.

We demonstrate by means of an example how onto constraints and path functional dependencies help to ensure that two schemas, one of which is the decomposition of the other, remain equivalent.

EXAMPLE 1

We suppose that a database designer is given the task to model the following scenario.

In a modern office, a phone belongs to the standard equipment. At the end of each month someone has to be charged for the accrued telecommunication costs. We assume that the office under consideration is in a university domain. So each faculty member is charged with the cost of at most one phone identified by its phone number. The bill is sent to the school a faculty member works for. Finally, schools are grouped into departments for better handling. For example each department is alloted a stock of phone numbers, preferably some with the same prefix.

In a first attempt to formalise this scenario, we conceive an object-oriented schema as depicted in Fig. 1. In this diagram, an arrow $c \xrightarrow{a} d$ *denotes that an attribute* a *is declared for the class* c *and that the type of this attribute is the class* d*, and an arrow* $c \dashrightarrow d$ *denotes that the class* c *is a subclass of the class* d*. So we have the classes* Phone-admin*,* Faculty*,* Person*, which is a superclass of the class* Faculty*,* School*,* Phone *and* Department*, and appropriate attributes declared for the class* Phone-admin*.*

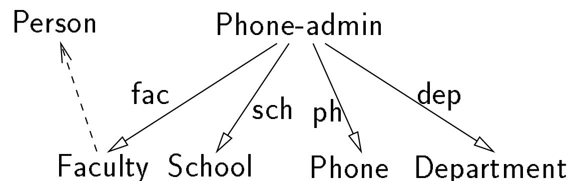


Figure 1: Simple object-oriented schema

In this formalisation the class Phone-admin *models the relationship among the entities mentioned above. Therefore we call such a class a* relationship class *[BMP96], and impose the restriction on its objects that all their attributes are defined.*

The restrictions on the data to be stored as mentioned in the text above can be formalised as path functional dependencies. The restriction "Each faculty member works for at most one school" is formalised by the path functional dependency Phone-admin(.fac $\rightarrow$ .sch)*. The semantics of this path functional dependency ensures that whenever two objects of the class* Phone-admin *agree on their value for the attribute* fac*, they agree on the*

value for the attribute sch*. In this case the name path functional dependency does not seem justified because the length of the path-functions* .fac *and* .sch *are only* 1*. Likewise we formalise the remaining restrictions and finally get the path functional dependencies*

$$\begin{aligned}&\mathsf{Phone\text{-}admin}(.\mathsf{fac} \rightarrow .\mathsf{sch})\ ,\\ &\mathsf{Phone\text{-}admin}(.\mathsf{fac} \rightarrow .\mathsf{ph})\ ,\\ &\mathsf{Phone\text{-}admin}(.\mathsf{sch} \rightarrow .\mathsf{dep})\ ,\ \textit{and}\\ &\mathsf{Phone\text{-}admin}(.\mathsf{ph} \rightarrow .\mathsf{dep})\ .\end{aligned}$$

Finally, the path functional dependency Phone-admin(.fac, .sch, .ph, .dep → .Id) *ensures that only one object exists for a value combination of the attributes* .fac*,* .sch*,* .ph *and* .dep*, i. e., the objects of the class* Phone-admin *simulate tuples.*

This first attempt is far from being complete for various reasons. It does not capture relationships like "A faculty member is charges with the cost of at most one phone". It does not cater for the recording of phone calls, and the subsequent billing. Nevertheless, the schema serves as starting point for further design steps.

Even in this present form the schema can be decomposed into a different one by means of pivoting. The outcome of this decomposition is the schema as depicted in Fig. 2.

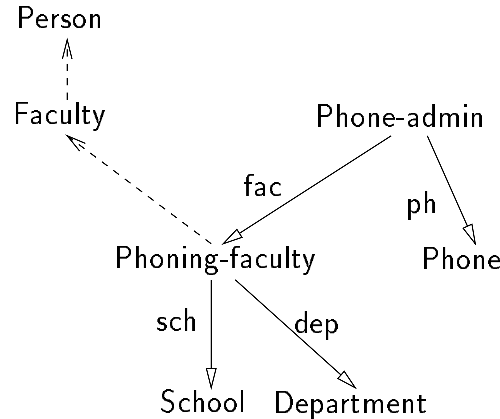


Figure 2: Pivoted object-oriented schema

In this transformation, we introduce a subclass Phoning-faculty*, which receives the attribute* sch *and* dep *from the class* Phone-admin*, and we change the semantic constraints to reflect the changes applied to the schema. Whenever one of the attributes* sch *and* dep *occurs in a path-function in any path functional dependency, we replace it by the path-function fragments* fac.sch *and* fac.dep*, respectively. So the path functional dependency* Phone-admin(.fac → .sch) *becomes in the new schema the path functional dependency* Phone-admin(.fac → .fac.sch)*, which demands that whenever two objects of the class* Phone-admin *agree on their value for attribute* fac*, they agree on the value delivered by the path-function* .fac.sch*.*

According to the inference rule simple prefix augmentation *(Def. 30), we derive this path functional dependency from the path functional dependency* Phoning-faculty(.Id →

.sch), *which trivially holds for the new schema due to the inference rule* simple attribution *(Def. 30). It is for this reason that we can discard the path functional dependency* Phone-admin(.fac → .fac.sch) *altogether. Likewise we drop other path functional dependencies or simplify them, thus we obtain the following set of semantic constraints:*

Phone-admin(.fac → .sch) *is discarded,*
Phone-admin(.fac → .ph) *remains unchanged* Phone-admin(.fac → .ph),
Phone-admin(.ph → .dep) *is changed into* Phone-admin(.ph → .fac.dep),
Phone-admin(.fac, .sch, .ph, .dep → .Id) *is reduced to* Phone-admin(.fac → .Id),
Phone-admin(.sch → .dep) *is changed into* Phoning-faculty(.sch → .dep),
and
we introduce Phone-admin{fac|Phoning-faculty}.

The last semantic constraint is an onto constraint, which secures that each object of the class Phoning-faculty *is referenced by some object of the class* Phone-admin *via the attribute* fac. *Since we want two equivalent schemas, we insert this onto constraint, because otherwise we obtained objects in the class* Phoning-faculty, *and so faculty members, which were members of some school and some department, but there were not a phone allocated to them. This were a state clearly not permissible in the original schema.*

While path functional dependencies are an extension of functional dependencies as known in the relational data model, we lack such correspondence in the case of onto constraints, although their nature is similar to inclusion dependencies. The relational data model separates the notion of a relation and of an attribute domain, whereas the object-oriented data model combines these two into the single notion of a class. An inclusion constraint operates on attribute domains, but translating the concept of onto constraints into the relational data model, we see that they work on relations and attribute domains, a combination simply not possible in the relational data model unless we extend it by the facility to access tuple identifiers [Cod79]. We make a similar observation when considering class inclusion constraints, which ensure that objects of one class are members of another class as well. The rationale for class inclusion constraints is that they are brought into the game by a combination of onto constraints and strict typing.

In Section 2, we introduce the simple data model, which is based on F-logic [KLW95], and so the data model retains the flexibility offered by F-logic and needed by us to incorporate the extensions used by us. Like the relational data model, it comes with a clear notion of a schema and an instance. Schemas encompass semantic constraints, and we concentrate on class inclusion constraints, onto constraints and path functional dependencies, which are introduced in Sect. 3. For theses classes of semantic constraints, we develop inference rules, and show them to be sound and complete. The proof of the completeness is only initially based on extensions of *C-Trees* and *Two-C-Trees* as used in the completeness proof for path functional dependencies alone [Wed92]. These constructs do not suffice due to the presence of onto constraints, because C-Trees and Two-C-Trees do not satisfy onto constraints. The general problem of decidability of the

logical implication remains open at this time, although the problem is positively solved for path functional dependencies alone [Wed92, IW94].

2 Data Model

Working in the field of database design calls for a high flexibility, a precise syntax and semantics of the underlying data model, because formalising desirable design-properties and finding syntactic characterisations involve the invention of new constructs like for example a new class of semantic constraints. From the vast number of object-oriented data models, we choose F-logic, because of its great flexibility, to define our data model. In this way, F-logic plays for us and our data model the same rôle as predicate logic for the relational data model.

2.1 F-Logic

We use a data model, which is based on F-logic as described by Kifer et. al. [KLW95]. To facilitate the access to this work, we present a concise description of the material used.

One of the basic notations of F-logic are *id-terms*, which are meant to denote *objects*, *classes* and *methods*. They are in our work only *constants* out of the set $\mathcal{F}$ and *variables* out of the set $\mathcal{V}$. By means of id-terms so called *molecular formulae* (*molecules* for short) are constructed. In the sequel C, D, O, A, V and R are id-terms. *Is-a assertions* of the form C::D (*class* is-a assertions) or of the form O:C (*object* is-a assertions) are molecules, which state that the class represented by the id-term C is a subclass of the class represented by the id-term D, and the object represented by the id-term O is element of the extension of the class represented by the id-term C. To alleviate the awkwardness that results from speaking of *id-terms representing something*, we will use throughout this paper the convention that we always speak of *id-terms* meaning sometimes even their interpretation, i. e. object, class or method, whenever it is clear from the context.

In addition there are *object molecules* of the form O[list of *method expressions*]. These method expressions are either *scalar data expressions* $A \rightarrow V$ or *scalar signature expressions* $A \Rightarrow R$. The intuitive meaning of a data expression $A \rightarrow V$ is that invoking the method A, which we call attribute, on the object O yields the value V. Likewise the signature expression $A \Rightarrow R$ should be understood as declaring the signature for the attribute A in the class O. In this case the type of the attribute value should be of the result type R.

Formulae are built up from simpler formulae by means of logical connectives and quantifiers. The simplest formulae are molecular formulae. If X is a variable, and ϕ and ψ are formulae, then so are $\phi \wedge \psi$, $\phi \vee \psi$, $\neg\phi$, $\exists X \phi$ and $\forall X \phi$. We use $\phi \longleftarrow \psi$ as another way of saying $\phi \vee \neg\psi$.

If any of the constructs defined above does not contain variables, the construct is said to be *ground*.

F-logic comes with a first-order model-theoretic semantics, which we do not want to present here in detail. In normal models, however, the connection between data expressions and their corresponding signature expressions[1] is missing in general. We introduce this connection by using *well-typedness axioms* (cf. Sect. 2.2) to restrict models to those that obey the typing declared by signature expressions. The details of these models are immaterial for the ensuing discussions (cf. [Pol00]).

Some more notations: The set of ground id-terms is denoted $U(\mathcal{F})$ (*Herbrand universe*), and the set of ground molecules is denoted $\mathcal{HB}(\mathcal{F})$ (*Herbrand base*). A subset of the Herbrand base closed under the logical entailment is an *H-structure*, which might be an *H-model* of a formula if the formula is a logical consequence (indicated as in classical logic by the symbol $\models$) of the H-structure. The *free* or *bound occurrence* of a variable in a formula is defined as in classical logic, hence a *sentence* is a formula without free occurrences of variables.

2.2 Schema and Instance

A data model describes two things, the actual data and components to give the data structure and to restrict the data to meaningful data.

We group together the components and call the result a *schema*, which we regard as being *time-invariant*. So a schema consists of the data structure description and *semantic constraints*. In an object-oriented data model we consider the *classes* identified in an application domain, their *hierarchy* and their *attributes* as relatively constant over the time.

Definition 2 (Schema)
A schema D *is of the form*

$$D = \langle \underbrace{\mathrm{CL}_D \cup \mathrm{AT}_D \cup \mathrm{HR}_D \cup \mathrm{SG}_D}_{\mathrm{ST}_D :=} | \mathrm{SC}_D \rangle$$

with the following finite sets

- *a set of "constants",* $\mathrm{CL}_D \subset \{t[\,] \mid t \in \mathcal{F}\}$*, for* class names*,*
- *a set of "constants",* $\mathrm{AT}_D \subset \{t[\,] \mid t \in \mathcal{F}\}$*, for* attribute names*,*
- *a set of ground class is-a assertions,* $\mathrm{HR}_D \subset \{c{::}d \mid c[\,], d[\,] \in \mathrm{CL}_D\}$*, to form the* class hierarchy*,*
- *a set of ground object molecules consisting only of scalar signature expressions,*

  $$\mathrm{SG}_D \subset \{c[a \Rightarrow r] \mid c[\,], r[\,] \in \mathrm{CL}_D, a[\,] \in \mathrm{AT}_D\} \ ,$$

  to declare signatures *for classes and attributes, and*

[1] They agree on their attribute and their arity.

- *a set of sentences,* SC_D*, as* semantic constraints *to restrict the set of allowed instances.*

The sets CL_D *of class names and* AT_D *of attribute names are disjoint,* $\mathrm{CL}_D \cap \mathrm{AT}_D = \emptyset$*.*

The classes in a schema form a hierarchy given by the set HR_D of class is-a assertions. On classes we declare attributes and their signatures by means of the set SG_D of object molecules with signature expressions. Due to the semantics of inheritance these signature declarations are inherited to subclasses. The set SC_D of sentences contains the semantic constraints, which restrict the data stored under a schema to meaningful data.

F-logic comes with no clear separation of the notions of an object, a class or an attribute. So by giving class names and attribute names we introduce such a distinction by ensuring that class names never occur at method or object positions and vice versa.

Class names and attribute names are technically not sets of constants but instead sets of trivial object molecules, nonetheless we will use these sets sometimes as if they were sets of constants.

Sometimes we add to a given schema D new semantic constraints Φ. We denote this addition by $D \cup \Phi$,

$$D \cup \Phi := \langle \mathrm{ST}_D | \mathrm{SC}_D \cup \Phi \rangle \quad .$$

As we usually want syntactically different things to be semantically different as well, we use *unique-name* axioms [Rei80] for this purpose. In addition, attributes should always return a value if the attributes are declared for an object. A fact that is ensured by the *not-null* axiom. The connection between signature expressions and data expressions is ensured by the presence of *well-typedness* axioms.

Definition 3 (Axioms)
The set AX *of* axioms *consists of three sets.*

- *The set of* unique-name *axioms*

$$\mathrm{UN} = \{ \overset{\circ}{\not\equiv} (t, t') \mid t, t' \in U(\mathcal{F}), t \not\equiv t' \} \quad .$$

- *The singleton set of the* not-null *axiom*

$$\mathrm{NN} = \{ \forall C \forall A \forall O \exists V \; O[A \to V] \longleftarrow C[A \Rightarrow (\,)] \wedge O{:}C \} \quad .$$

- *The set of* well-typedness *axioms*

$$\begin{aligned} \mathrm{WT} = & \{ \forall O \forall A \forall V \exists C \;\; (C[A \Rightarrow (\,)] \wedge O{:}C) \longleftarrow O[A \to V] \} \\ & \cup \\ & \{ \forall O \forall A \forall V \forall C \forall R \;\; V{:}R \longleftarrow C[A \Rightarrow R] \wedge O{:}C[A \to V] \} \quad . \end{aligned}$$

When the class hierarchy is cyclic, for example we can imply $c{::}d$ and $d{::}c$ from the set HR_D, the semantics of F-logic considers these classes c and d as identical, $\mathrm{HR}_D \models c \overset{\circ}{=} d$,

and hence a violation of the unique-name axioms. This is a feature we do not desire, and therefore we demand that the class hierarchy is acyclic througout this work by using only schemas such that $\mathrm{HR}_D \models \mathrm{UN}$ holds.

Later on we are often interested in the set of attributes declared for a class and in attributes that are declared for exactly one class.

Definition 4 ((Proper) Attribute)
Let D be a schema.

- *The set* $\{a \mid \mathrm{HR}_D \cup \mathrm{SG}_D \models c[a \Rightarrow (\,)]\}$ *of attributes for a class c is denoted* $\mathrm{At}_D(c)$.
- *An attribute $a \in \mathrm{AT}_D$ is called a* proper attribute *for a class $c \in \mathrm{CL}_D$:iff* $\mathrm{SG}_D \models c[a \Rightarrow (\,)]$ *and* $\mathrm{SG}_D \not\models d[a \Rightarrow (\,)]$ *holds for all classes $d \in \mathrm{CL}_D$ with $d \not\equiv c$.*

In the definition above, we use the logical entailment to imply knowledge about the class hierarchy and the signature declarations. Without a proof we note that the logical entailment for is-a assertions and signature atoms is decidable for a finite set of is-a assertions and signature atoms.

Having described the part of our data model that is relatively stable over time, we come to the part, which is subject to frequent changes over time, the data itself, which is the objects populating classes and their attribute values. Both aspects are formalised as formulae, which we separate according to their nature, object is-a assertions for class populations and scalar data atoms for attribute value definitions.

Definition 5 (Extension)
An extension $f \in \mathrm{ext}(D)$ *of a schema D is of the form* $f = \langle \mathrm{pp}_f | \mathrm{ob}_f \rangle$ *with the following sets*

- *a set of ground object is-a assertions,*

$$\mathrm{pp}_f \subset \{o{:}c \mid o \in \mathcal{F}\backslash(\mathrm{CL}_D \cup \mathrm{AT}_D), c \in \mathrm{CL}_D\} \enspace ,$$

  populating classes*, and*

- *a set of ground object molecules with only scalar data expressions,*

$$\mathrm{ob}_f \subset \{o[a \rightarrow v] \mid o, v \in \mathcal{F}\backslash(\mathrm{CL}_D \cup \mathrm{AT}_D), a \in \mathrm{AT}_D\} \enspace ,$$

  for the definition of attribute values *for objects.*

We do not permit to define new classes, changes to the class hierarchy, or the introduction of new signatures. We bind id-terms in extensions to constants to obviate the deduction of knowledge from the internal structure of id-terms. Additionally, we want a clear separation between the domains of objects, classes and attributes, which are interwoven in F-logic.

A *completion* of an extension under a schema comprises all that can be deduced from the extension and the schema, in this case ground molecules.

Definition 6 (Completion)
Let f be an extension of a schema D. The completion *of the extension f under the schema D is*

$$\mathrm{compl}_D(f) := \{\tau \in \mathcal{HB}(\mathcal{F}) \mid \mathrm{ST}_D \cup \mathrm{pp}_f \cup \mathrm{ob}_f \models \tau\} \ .$$

Completions are the smallest H-models of the union $\mathrm{ST}_D \cup \mathrm{pp}_f \cup \mathrm{ob}_f$.

An instance is an extension of a schema that obeys the semantic constraints given in the schema and that guarantees that each object and value in attribute-value definitions are members of some class. In addition, instances or better their completions must satisfy unique-name, not-null and well-typedness axioms.

Definition 7 (Instance)
Let f be an extension of a schema D. Then the extension f is called a(n) (allowed) instance *of the schema D, $f \in \mathrm{sat}(D)$, if $\mathrm{compl}_D(f)$ is an H-model of the axioms* AX *and the semantic constraints* SC_D.

Due to the limitations given for instances, we determine the set of objects in an instance solely be means of the set pp_f of object is-a assertions. To determine the classes an object is element of in an instance it is sufficient to know the class hierarchy (HR_D) and the class population (pp_f).

Definition 8 (Objects and Class-Labels)
Let f be an extension of a schema D.

- *The set of all objects in the extension f is denoted* $\mathrm{obj}(f) := \{o \mid ex.\ c\colon o{:}c \in \mathrm{pp}_f\}$.
- *The* class-label $\lambda_{\mathrm{Cl}}(o)$ *for an object $o \in \mathrm{obj}(f)$ is the set of classes object o is element of,* $\lambda_{\mathrm{Cl}}(o) := \{c \mid \mathrm{HR}_D \cup \mathrm{pp}_f \models o{:}c\}$.

So far semantic constraints can be any sentences. A fact we will change in the next section after having introduced classes of semantic constraints.

3 Semantic Constraints

In the relational data model there are a number of well-understood classes of semantic constraints like functional dependencies, multi-valued dependencies, join dependencies or inclusion dependencies. Such commonly known and deeply investigated classes of semantic constraints are missing in object-oriented data models. We surmise that the root of object-oriented data models in object-oriented programming accounts for this defect. The flexibility of expressing arbitrary semantic constraints in a programming language leads to burden the application programmer with the specification and implementation of semantic constraints instead of leaving the “implementation” to the database management system. In that way the principle of *integrity independence* [Cod90] is violated.

We take a different approach by defining three classes of semantic constraints, *class inclusion constraints*, *onto constraints* and *path functional dependencies*. The rôle of class inclusion dependencies is only a subordinate part. They are needed for the completeness of the inference rules for onto constraints and path functional dependencies.

3.1 Class Inclusion and Onto Constraints

A class inclusion constraint $c \subset d$ demands that every object of the class c is a member of the class d as well.

DEFINITION 9 (CLASS INCLUSION CONSTRAINT)

- *A* class inclusion constraint *for classes* $c, d \in \mathrm{CL}_D$ *over a schema* D *is of the form* $c \subset d$.
- *The* class inclusion constraint formula *for a class inclusion constraint* $c \subset d$ *is* $\forall O \;\; O{:}d \longleftarrow O{:}c$.
- *The symbol* CIC_D *denotes the set of all class inclusion constraint formulae in the set* SC_D *of semantic constraints.*

Although the second sort of semantic constraints can play a rôle of its own, we introduce them mainly in connection to the transformation *pivoting* [BMPS96], because onto constraints ensure the equivalence between an original schema and the transformed schema. An onto constraint $c\{a|d\}$ demands that every object of the class d is referenced by an object of the class c via the attribute a.

DEFINITION 10 (ONTO CONSTRAINT)

- *An* onto constraint *for a class* $c \in \mathrm{CL}_D$ *over a schema* D *is of the form* $c\{a|d\}$, *where* $a \in \mathrm{At}_D(c)$ *is an attribute and* $d \in \mathrm{CL}_D$ *is a class.*
- *The* onto constraint formula *for an onto constraint* $c\{a|d\}$ *is*

$$\forall V \exists O \;\; (O{:}c[a \rightarrow V] \wedge c[a \Rightarrow (\,)]) \longleftarrow V{:}d \;\; .$$

- *The set of onto constraint formulae for a class* c *in the set* SC_D *of semantic constraints is denoted* $\mathrm{OC}_D(c)$. *The symbol* OC_D *denotes the set of all onto constraint formulae in the set* SC_D *of semantic constraints.*

Often we blur the clear distinction between syntax and semantics and write $c\{a|d\} \in \mathrm{OC}_D(c)$ whenever that is appropriate.

EXAMPLE 11

In the second pivoted schema in Exam. 1, we find the onto constraint

$$\textsf{Phone-admin}\{\textsf{fac}|\textsf{Phoning-faculty}\} \;\; ,$$

which ensures in an instance that each element of the class Phoning-faculty *is an attribute value for the attribute* fac *of some object of the class* Phone-admin.

The symbol Υ_D denotes the set of class inclusion constraint and onto constraint formulae in a schema D,

$$\Upsilon_D := \mathrm{CIC}_D \cup \mathrm{OC}_D \;\; .$$

We now give inference rules for class inclusion constraints and onto constraints without introducing path functional dependencies, because path functional dependencies have no influence on the derivation of class inclusion constraints and onto constraints.

The first rule incarnates the transitive nature of class inclusion constraints. The second rule captures the impact of the class hierarchy: if a class is a subclass of another class, each element of the subclass is an element of its superclass. As the class hierarchy depends solely on the underlying schema, this rule introduces a dependency of the inference mechanism on the underlying schema. The next rule embodies the interplay between onto constraints and signature declarations and is the cause for the presence of class inclusion constraints. The effect is the introduction of class inclusion constraints orthogonal to the class hierarchy. While the last three rules saw class inclusion constraints as outcome, the next two rules deal with the inference of onto constraints. The first of these rules allows a restriction of the range of an onto constraint, whereas the second grants the relaxation of the domain of an onto constraint.

Definition 12 (Inference Rules for CICs and OCs)
Let v be a class inclusion constraint or onto constraint over a schema D. The constraint v is derivable from the set Υ_D, written $\Upsilon_D \vdash_D v$, :iff it is a member of the set Υ_D or the result of one or more applications of the following inference rules.

I1. Class inclusion transitivity: *If both $c \subset c'$ and $c' \subset c''$ can be derived, then so can $c \subset c''$.*

I2. Subclass inclusion: *If $\mathrm{HR}_D \models c{::}c'$ holds, derive $c \subset c'$.*

I3. Signature inclusion: *If $d\{a|c\}$ can be derived, where $\mathrm{HR}_D \cup \mathrm{SG}_D \models d[a \Rightarrow c']$, then so can $c \subset c'$.*

I4. Range restriction: *If $c \subset c'$ and $d\{a|c'\}$ can be derived, then so can $d\{a|c\}$.*

I5. Domain relaxation: *If $c \subset c'$ and $c\{a|d\}$ can be derived, where $a \in \mathrm{At}_D(c')$, then so can $c'\{a|d\}$.*

In this definition we bring the logical entailment to bear to extract knowledge about the class hierarchy and the signature declarations; but we keep in mind that these extractions can be algorithmically decided.

We demonstrate how the inference rules work by means of an example, which serves as running example throughout this work. The purpose of this example is to convey an intuition for the technicalities involved in the completeness proof of the inference rules for class inclusion constraints, onto constraints and path functional dependencies. Therefore we do not come up with an example that offers an interpretation in the real world rather it uses class names and attribute names like e', e, k, l and alike.

Example 13
The example is based on a schema, which includes at this point only onto constraints. Class inclusion constraints come into play by derivation, whereas path functional dependencies are not considered at this point. We add them later on after having introduced them.

The schema S we use consists of the components

$$\begin{aligned}
\mathrm{CL}_S &= \{e', e, f'', f', f, g, h, i, j\} \ , \\
\mathrm{AT}_S &= \{k, l, m, n, o, p, q, r\} \ , \\
\mathrm{HR}_S &= \{e'{::}e, f''{::}f'\} \ , \\
\mathrm{SG}_S &= \{e[m \Rightarrow h], e'[l \Rightarrow f], \\
&\quad f[n \Rightarrow g; o \Rightarrow g], f''[k \Rightarrow e'], \\
&\quad h[p \Rightarrow i; q \Rightarrow j], \\
&\quad i[r \Rightarrow h]\} \ , \ and \\
\mathrm{SC}_S &= \{e'\{m|h\}, e'\{l|f'\}, f''\{k|e'\}\} \ .
\end{aligned}$$

Its schema is depicted in Fig. 3. An interesting onto constraint in this schema is the

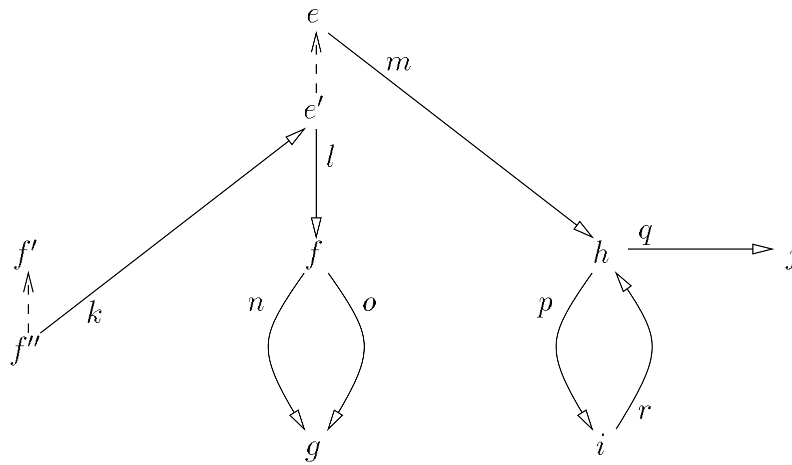


Figure 3: The schema in the running example

constraint $e'\{l|f'\}$*, which has a deep impact on the behaviour of instances. The range class* f' *is not a result class of the attribute* l *declared on the class* e' *nor a subclass of any result class. But there exists a signature declaration* $e'[l \Rightarrow f]$*. So every object of the class* f' *must be referenced by an object of the class* e' *via the attribute* l *due to the onto constraint* $e'\{l|f'\}$ *and the object of the class* f' *must be element of the class* f *to satisfy the well-typedness axioms. This behaviour is captured by the inference rule I3 signature inclusion.*

An instance of the schema S is

$$\begin{aligned}
s = \langle\, & \{o_e{:}e', o_f{:}f'', o_f{:}f, o_g{:}g, o_h{:}h, o_i{:}i, o_j{:}j\} \\
& | \\
& \{o_e[m \to o_h; l \to o_f], \\
& \ o_f[n \to o_g; o \to o_g; k \to o_e], \\
& \ o_h[p \to o_i; q \to o_j], \\
& \ o_i[r \to o_h]\} \,\rangle \ .
\end{aligned}$$

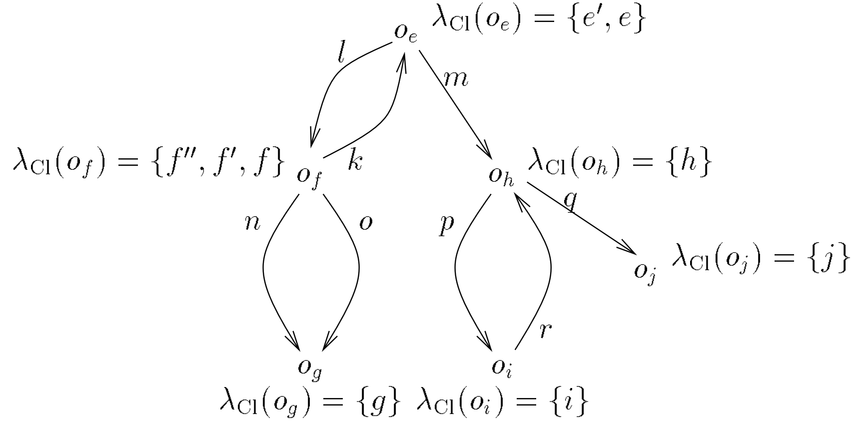


Figure 4: An instance of the schema in the running example

This instance is depicted in Fig. 4.

We use the semantic constraints in the schema S to exhibit how the inference rules I1 to I5 for class inclusion constraints and onto constraints work. We give for every inference rule one example in Fig. 5. To summarise the set of all class inclusion constraints

$$\left.\begin{array}{l} e'\{l|f'\} \\ e'[l \Rightarrow f] \end{array}\right\} \overset{\mathrm{I3}}{\vdash_S} f' \subset f$$

$$f''::f'\} \overset{\mathrm{I2}}{\vdash_S} f'' \subset f'$$

$$\left.\begin{array}{l} f'' \subset f' \\ f' \subset f \end{array}\right\} \overset{\mathrm{I1}}{\vdash_S} f'' \subset f$$

$$\left.\begin{array}{l} f'' \subset f' \\ e'\{l|f'\} \end{array}\right\} \overset{\mathrm{I4}}{\vdash_S} e'\{l|f''\}$$

$$\left.\begin{array}{l} e' \subset e \\ e'\{m|h\} \\ e[m \Rightarrow h] \end{array}\right\} \overset{\mathrm{I5}}{\vdash_S} e\{m|h\}$$

Figure 5: Application of the inference rules I1 to I5

derivable from the set SC_S of semantic constraints under the schema S is

$$\{e' \subset e', e \subset e, f'' \subset f'', f' \subset f', f \subset f, g \subset g, h \subset h, i \subset i, j \subset j,$$
$$e' \subset e, f'' \subset f', f'' \subset f, f' \subset f\} \text{ , and}$$

the set of all onto constraints we can derive from the set SC_S of semantic constraints

under the schema S is

$$\{e'\{m|h\}, e\{m|h\},\\ e'\{l|f'\}, e'\{l|f''\},\\ f''\{k|e'\}\}\ .$$

The instance s satisfies the semantic constraints and due to the correctness of the inference rules, which we show in Theor. 33, also all derivable constraints.

The *closure* of the set of class inclusion constraints and onto constraints in a schema is exactly the set of all class inclusion constraints and onto constraints derivable from this very set of class inclusion constraints and onto constraints.

Definition 14 (Closure for CICs and OCs)
The closure *of the set Υ_D, written $\Upsilon_D^{+_D}$, is the set of all class inclusion constraints and onto constraints υ over the schema D, where $\Upsilon_D \vdash_D \upsilon$.*

3.2 Way-Descriptions and Path-Functions

In preparation to define path functional dependencies, we need the concept of a path or better of a path-function. But instead of simply defining path-functions, we introduce the more general notion of a *way-description*, because we use this generalisation of path-functions extensively in the completeness proof. The basic idea underlying these completeness proofs[2] is to construct a counterexample that satisfies the semantic constraints declared in the schema D but does not satisfy a semantic constraint not derivable from the other semantic constraints. So as to prove the latter property, we construct this counterexample based on the knowledge of what can be derived and not on the semantic notion of what can be logically implied.

The construction of the counterexample is iterative. We begin with a core counterexample, which partly satisfies the semantic constraints (class inclusion constraints and path functional dependencies) and does not satisfy a not-derivable constraint. Now whenever we encounter an object in the construction process that violates an onto constraint, we remedy this violation. For example, let $c\{a|d\}$ be an onto constraint of a schema and o be an object of the class d in the counterexample, which is not referenced by any object of the class c via the attribute a. In this case we cure the violation by introducing a new object of the class c that then references the object o via the attribute a.

In this iterative construction process we keep track of the position of newly inserted objects relative to certain objects, called *roots*. We accomplish this tracking by remembering the attributes we follow when trying to reach newly inserted objects from roots by traversing the attributes in the graph spanned by an instance. But sometimes we follow edges in the reverse direction; so instead of following simple paths we follow permissible ways.

[2] We divide the completeness proof into several smaller pieces.

A *folder* of a class is now the first concept based on the principle of using "derived knowledge". The folder of a class specifies all classes an object of the class must be an element of according to the semantic constraints and the class hierarchy.

DEFINITION 15 (FOLDER)
We define the folder *of a class* $c \in \mathrm{CL}_D$ *as* $\mathrm{Folder}_D(c) := \{d \mid \Upsilon_D \vdash_D c \subset d\}$.

EXAMPLE 16
The folder of the class f'' *from the schema* S *in Exam. 13 is*

$$\mathrm{Folder}_S(f'') = \{f'', f', f\} \ .$$

The core counterexample is a proper instance. So having determined which classes the root objects have to be a member of, we find out about the class memberships of other objects included in the initial instance due to the not-null axiom and well-typedness axioms, and later on in the iterated instances even due to unsatisfied onto constraints. For this task we employ *way-descriptions*, which describe an *attribute-value way* in an instance. An instance can be seen as a labelled graph; each object constitutes a node with a label that comprises the classes an object is a member of, and a directed edge between two nodes (objects) with a label that signifies that the source object holds the target object as value for the attribute that is given by the edge-label. Way-descriptions are an extension of *path-functions* [Wed89, Wed92], which capture that we follow attribute-value edges like $o \xrightarrow{a} o'$ not only in the forward direction from the object o to the object o' but also in the reverse direction from the object o' to the object o.

Syntactically, way-descriptions are strings of attribute names separated by dots and minuses, where dots mean the forward traversal while minuses signify the backward traversal. In our first definition of *arbitrary way-descriptions*, we arbitrarily string attribute names together.

DEFINITION 17 (ARBITRARY WAY-DESCRIPTION AND PATH-FUNCTIONS)

- *The set of* arbitrary way-descriptions $\mathrm{W}_{\mathrm{arb}}(D)$ *over a schema* D *consists of all finite sequences of 'dot or minus separated' attribute names together with the* identity way description *denoted as* .Id *where* $\mathrm{Id} \notin \mathcal{V} \cup \mathcal{F}$.

$$\mathrm{W}_{\mathrm{arb}}(D) := \{\pi_1 m_1 \cdots \pi_n m_n \mid n > 0 \text{ and } m_i \in \mathrm{AT}_D, \pi_i \in \{., -\}, i \in \{1, \ldots, n\}\} \cup \{.\mathrm{Id}\}$$

- *The length of the way-description* .Id *is* 0, $\mathrm{len}(.\mathrm{Id}) := 0$. *The length of a way-description* $w \equiv \pi_1 m_1 \cdots \pi_n m_n$ *is* n, $\mathrm{len}(w) := n$.

- *The set of* arbitrary path-functions $\mathrm{P}_{\mathrm{arb}}(D) \subset \mathrm{W}_{\mathrm{arb}}(D)$ *over a schema* D *consists of all arbitrary way-descriptions with only 'dots' occurring in them.*

EXAMPLE 18
For the schema S *in Exam. 13 the following are arbitrary way-descriptions over the schema* S*:*

$$-k-m.p, -k.p.r, .\mathrm{Id}, .r.m.k, .k.m, .m.k, -m.k \text{ and } -m-k \ .$$

In the set above the arbitrary way-descriptions $.\mathrm{Id}$*,* $.r.m.k$*,* $.k.m$ *and* $.m.k$ *are arbitrary path-functions.*

The set of arbitrary way-descriptions is too large, because way-descriptions describe attribute-value ways that do not appear in any instance of the underlying schema. In addition, there are arbitrary way-descriptions we are not interested in. These way-descriptions take on the form $\cdots .a{-}a \cdots$ or $\cdots {-}a.a \cdots$, which we exclude by means of the concatenation of arbitrary way-descriptions.

Definition 19 (Concatenation of Arbitrary Way-Descriptions)
Let $w_1, w_2 \in \mathrm{W}_{\mathrm{arb}}(D)$ *be two arbitrary way-descriptions over a schema* D*. Then the concatenation, written* $w_1 \circ w_2$*, is defined as*

$$w_1 \circ w_2 := \begin{cases} .\mathrm{Id} & \text{if } w_1 = w_2 = .\mathrm{Id}, \text{ or } w_1 = \pi m,\ w_2 = \bar{\pi} m \\ w_1 & \text{if } w_1 \neq .\mathrm{Id},\ w_2 = .\mathrm{Id} \\ w_2 & \text{if } w_1 = .\mathrm{Id},\ w_2 \neq .\mathrm{Id} \\ w_1' & \text{if } w_1 = w_1' \pi m,\ w_2 = \bar{\pi} m \\ w_2' & \text{if } w_1 = \pi m,\ w_2 = \bar{\pi} m w_2' \\ w_1' \circ w_2' & \text{if } w_1 = w_1' \pi m,\ w_2 = \bar{\pi} m w_2' \\ w_1 w_2 & \text{otherwise} \end{cases} ,$$

where, if $\pi = .$ *or* $\pi = -$*, then* $\bar{\pi} = -$ *or* $\bar{\pi} = .$*, respectively.*

Example 20
We give examples for the concatenation of arbitrary way-descriptions for the arbitrary way-descriptions in Exam. 18.

$$\begin{aligned} .\mathrm{Id} \circ .\mathrm{Id} &= .\mathrm{Id} \\ .k.m \circ .m.k &= .k.m.m.k \\ .k.m \circ -m.k &= .k.k \\ .k.m \circ -m-k &= .\mathrm{Id} \end{aligned}$$

The concatenation is associative as the next lemma indicates.

Lemma 21 (Associativity of Arbitrary Way-Descriptions)
Let $w_1, w_2, w_3 \in \mathrm{W}_{\mathrm{arb}}(D)$ *be three arbitrary way-descriptions over a schema* D*. The concatenation of arbitrary way-descriptions is associative,*

$$(w_1 \circ w_2) \circ w_3 = w_1 \circ (w_2 \circ w_3) \ .$$

Proof. The proof of this lemma includes some technical subtleties but is obvious.

□

Our objective is to use only those way-descriptions, *well-formed way-descriptions*, that always describe an attribute-value way starting at an object in certain instances. These instances are minimal in the sense that apart from a given set of objects they only contain objects whose existence is demanded by the semantic constraints and axioms,

and they form a directed acyclic graph. Then we intend to determine the compulsory class memberships of the objects reached by attribute-value ways.

The definition of a well-formed way-description starts with the shortest possible well-formed way-description, .Id, and then considers longer descriptions by distinguishing the prolongation of already existing well-formed way-descriptions by an attribute either in forward or reverse direction.

We simultaneously define the *domain* and the *range* of a well-formed way-description. The former is the set of classes from which a way-description may start, i. e., a well-formed way-description always describes some attribute-value way if the starting point is an object that is element of a *domain class* (a class in the *domain of the way-description*). The latter is a set of classes an end node of an attribute-value way must be a member of.

DEFINITION 22 ((WELL-FORMED) WAY-DESCRIPTION)

- *The set of* (well-formed) way-descriptions $\mathrm{W_{wf}}(D) \subset \mathrm{W_{arb}}(D)$ *over a schema* D *is the smallest set such that*
  - $.\mathrm{Id} \in \mathrm{W_{wf}}(D)$, *where*
    - $\mathrm{Dom}_D(.\mathrm{Id}) := \mathrm{CL}_D$, *and*
    - $\mathrm{Ran}_D(c, .\mathrm{Id}) := \mathrm{Folder}_D(c)$ *for all* $c \in \mathrm{CL}_D$,
  - *if* $w \in \mathrm{W_{wf}}(D)$ *is a (well-formed) way-description with a domain class* $c \in \mathrm{Dom}_D(w)$ *such that* $a \in \bigcup_{d \in \mathrm{Ran}_D(c,w)} \mathrm{At}_D(d)$ *for some attribute* a *and* $\mathrm{len}(w) < \mathrm{len}(w \circ .a)$, *then* $w \circ .a \in \mathrm{W_{wf}}(D)$, *where*

$$\mathrm{Dom}_D(w \circ .a) := \{e \in \mathrm{Dom}_D(w) \mid a \in \bigcup_{f \in \mathrm{Ran}_D(e,w)} \mathrm{At}_D(f)\} \text{ , and}$$

$$\mathrm{Ran}_D(e, w \circ .a) := \{h \in \mathrm{Folder}_D(g) \mid \text{ex. } f\colon\ f \in \mathrm{Ran}_D(e,w) \text{ and } \mathrm{HR}_D \cup \mathrm{SG}_D \models f[a \Rightarrow g]\}$$

$$\text{for all } e \in \mathrm{Dom}_D(w \circ .a) \text{ ,}$$

  - *if* $w \in \mathrm{W_{wf}}(D)$ *is a (well-formed) way-description with a domain class* $c \in \mathrm{Dom}_D(w)$ *such that* $d\{a|b\} \in \Upsilon_D^{+_D}$ *for some class* $b \in \mathrm{Ran}_D(c, w)$, *and* $\mathrm{len}(w) < \mathrm{len}(w \circ -a)$, *then* $w \circ -a \in \mathrm{W_{wf}}(D)$, *where*

$$\mathrm{Dom}_D(w \circ -a) := \{e \in \mathrm{Dom}_D(w) \mid \text{ex. } f, g\colon\ f \in \mathrm{Ran}_D(e,w) \text{ and } g\{a|f\} \in \Upsilon_D^{+_D}\} \text{ , and}$$

$$\mathrm{Ran}_D(e, w \circ -a) := \{h \in \mathrm{Folder}_D(g) \mid \text{ex. } f\colon\ f \in \mathrm{Ran}_D(e,w) \text{ and } g\{a|f\} \in \Upsilon_D^{+_D}\}$$

$$\text{for all } e \in \mathrm{Dom}_D(w \circ -a) \text{ .}$$

- *For each class* $c \in \mathrm{CL}_D$ *we write* $\mathrm{WayDes}_D(c)$ *to denote all well-formed way-descriptions* $w \in \mathrm{W_{wf}}(D)$ *where* $c \in \mathrm{Dom}_D(w)$, *i. e. all way-descriptions starting at the class* c.

- *The set* $\mathrm{PathFuncs}_D(c) \subset \mathrm{WayDes}_D(c)$ *denotes the set of all well-formed path-functions starting at the class* c.

When we speak in the sequel of way-descriptions (path-functions), we always mean well-formed way-descriptions (path-functions) unless otherwise mentioned.

Example 23

Some of the arbitrary way-descriptions in Exam. 18 are well-formed. First of all the arbitrary way-description $.\mathrm{Id}$ *is well-formed with*

$$\begin{aligned} \mathrm{Dom}_S(.\mathrm{Id}) &= \mathrm{CL}_S = \{e', e, f'', f', f, g, h, i, j\}, \quad \textit{and} \\ \mathrm{Ran}_S(f'', .\mathrm{Id}) &= \mathrm{Folder}_S(f'') = \{f'', f', f\} \ . \end{aligned}$$

Based on this knowledge, we calculate the domains and some ranges for the way-descriptions $.k.m$ *and* $-m$. *We begin with the way-description* $.k.m$. *Therefore we show first that the way-description* $.k$ *is well-formed and then determine its domain and range for some domain class.*

We already know that the way-description $.\mathrm{Id}$ *is well-formed. Following the definition of way-descriptions in the case of a forward prolongation, we derive that for the class* $f'' \in \mathrm{Dom}_S(.\mathrm{Id})$ *the range* $\mathrm{Ran}_S(f'', .\mathrm{Id})$ *of the way-description* $.\mathrm{Id}$ *is* $\mathrm{Folder}_S(f'') = \{f'', f', f\}$. *The set* $\mathrm{At}_S(f'')$ *of attributes for the class* f'' *is* $\{k\}$, *hence* $k \in \mathrm{At}_S(f'')$. *Finally, the inequation* $\mathrm{len}(.\mathrm{Id}) = 0 < 1 = \mathrm{len}(.k) = \mathrm{len}(.\mathrm{Id} \circ .k)$ *holds, and therefore the way-description* $.k$ *is well-formed with*

$$\begin{aligned} \mathrm{Dom}_S(.k) &= \mathrm{Dom}_S(.\mathrm{Id} \circ .k) \\ &= \{a \in \mathrm{Dom}_S(.\mathrm{Id}) \mid k \in \bigcup_{b \in \mathrm{Ran}_S(a, .\mathrm{Id})} \mathrm{At}_S(b)\} \\ &= \{f''\} \end{aligned}$$

and

$$\begin{aligned} \mathrm{Ran}_S(f'', .k) &= \mathrm{Ran}_S(f'', .\mathrm{Id} \circ .k) \\ &= \{c \in \mathrm{Folder}_S(b) \mid \textit{ex. } a\colon a \in \mathrm{Ran}_S(f'', .\mathrm{Id}) \textit{ and} \\ &\qquad\qquad \mathrm{HR}_S \cup \mathrm{SG}_S \models a[k \Rightarrow b]\} \\ &= \{e', e\} \end{aligned}$$

Now we know $f'' \in \mathrm{Dom}_S(.k)$, $e' \in \mathrm{Ran}_S(f'', .k)$, $m \in \mathrm{At}_S(e')$ *and* $\mathrm{len}(.k) = 1 < 2 = \mathrm{len}(.k.m) = \mathrm{len}(.k \circ .m)$, *and thus the way-description* $.k.m$ *is well-formed with*

$$\begin{aligned} \mathrm{Dom}_S(.k.m) &= \{f''\} \ , \ \textit{and} \\ \mathrm{Ran}_S(f'', .k.m) &= \{h\} \ . \end{aligned}$$

We exhibit how the definition of way-descriptions is used with way-descriptions like $-m$. *The way-description* $-m$ *is well-formed, because* $h \in \mathrm{Dom}_S(.\mathrm{Id})$, $h \in \mathrm{Ran}_S(h, .\mathrm{Id})$, $e'\{m|h\} \in \mathrm{SC}_S$ *and* $\mathrm{len}(.\mathrm{Id}) < \mathrm{len}(-m) = \mathrm{len}(.\mathrm{Id} \circ -m)$ *holds. So the domain is*

$$\mathrm{Dom}_S(-m) = \{h\}$$

and the range is

$$\mathrm{Ran}_S(h, -m) = \{e', e\} \ .$$

When we concatenate two way-descriptions such that the second is applicable to a class in the range of the first, we obtain a way-description again, independent of the form of both.

LEMMA 24 (CONCATENATION OF WAY-DESCRIPTIONS)
Let $w, w' \in \mathrm{W_{wf}}(D)$ be two way-descriptions over a schema D such that there exist classes c, c' with $c \in \mathrm{Dom}_D(w)$ and $c' \in \mathrm{Ran}_D(c, w) \cap \mathrm{Dom}_D(w')$. Then $w \circ w'$ is a way-description over the schema D, $w \circ w' \in \mathrm{W_{wf}}(D)$, with $c \in \mathrm{Dom}_D(w \circ w')$ and $\mathrm{Ran}_D(c', w') \subset \mathrm{Ran}_D(c, w \circ w')$.

PROOF. The proof is by induction on the length of the way-description w'.

□

A well-formed way-description might describe several attribute-value ways when starting at an object. Mostly we are interested in the end nodes of an attribute-value way described by a well-formed way-description.

DEFINITION 25 (APPLICATION OF WAY-DESCRIPTIONS)
Let f be an instance of a schema D, $o \in \mathrm{obj}(f)$ be an object, $c \in \lambda_{\mathrm{Cl}}(o)$ be a class, and $w \in \mathrm{WayDes}_D(c)$ be a way-description.

- *If $w = .\mathrm{Id}$, then $o.w = \{o\}$.*
- *If $w = w' \circ .a$, then $o.w = \{o'' \mid \text{ex. } o' \colon o' \in o.w' \text{ and } \mathrm{ob}_f \models o'[a \to o'']\}$.*
- *If $w = w' \circ -a$, then $o.w = \{o'' \mid \text{ex. } o' \colon o' \in o.w' \text{ and } \mathrm{ob}_f \models o''[a \to o']\}$.*

EXAMPLE 26
The application of the way-descriptions $.k$, $.k.m$, $-m$ and $-m-k$ to some objects in the extension s in Exam. 13 yields the following results

$$\begin{array}{ll} o_f..k = & \{o_e\} \ , \\ o_f..k.m = & \{o_h\} \ , \\ o_h.-m = & \{o_e\} \ , \ \textit{and} \\ o_h.-m-k = & \{o_f\} \ . \end{array}$$

3.3 Path Functional Dependencies

Path-functions are a generalisation of attributes as known in the relational data model. *Path functional dependencies* as introduced by Weddell [Wed89, Wed92] are a generalisation of functional dependencies as known in the relational data model. While functional dependencies are defined on a relation scheme, path functional dependencies are declared for classes and employ path-functions in lieu of simple attributes in their left-hand and right-hand sides. A path functional dependency then states if two objects in the class given in the dependency deliver identical results for each path-function in the left side of the dependency, the objects have to agree on the value for each path-function in the right side.

Example 27
In the motivating Exam. 1 in the introduction, we find the path functional dependency

$$\text{Phone-admin}(.\text{ph} \rightarrow .\text{fac.dep}) \ ,$$

which formalises the fact that each department is alloted a stock of phone numbers, preferably some with the same prefix. So whenever two objects of the class Phone-admin *agree on the value for the phone number, the faculty member that is charged with the telecommunications costs works in exactly one department.*

The path-functions in a path functional dependency are chosen in a way that they start all at the class the path functional dependency is declared for. In that way, we guarantee when we apply one of these path-functions to an object of the respective class, we always obtain a defined result.

We have to bear in mind here that the set of path-functions starting at a class is enlarged by class inclusion constraints and onto constraints. When a class inclusion constraint $c \subset d$ holds, every path-function or better way-description starting at the class d starts also at the class c.

Definition 28 (Path Functional Dependency)

- *A* path functional dependency *for a class* $c \in \mathrm{CL}_D$ *over a schema* D *is of the form*

$$c(p_1 \cdots p_k \rightarrow p_{k+1} \cdots p_n) \ ,$$

where $k \in \{1, \ldots, n-1\}$ *and* $p_i \in \mathrm{PathFuncs}_D(c)$ *for* $i \in \{1, \ldots, n\}$.

- *To define that a database instance satisfies a path functional dependency, we construct formulae. We facilitate this task by using a term* $X[p \rightarrow Y]$ *with*

$$p \equiv .m_1^p \cdots .m_l^p \in \mathrm{P_{arb}}(D) \backslash \{.\mathrm{Id}\}$$

as a shorthand form for

$$(\exists X_1^p \cdots \exists X_{l-1}^p (X[m_1^p \rightarrow X_1^p] \wedge X_1^p[m_2^p \rightarrow X_2^p] \wedge \cdots \wedge X_{l-1}^p[m_l^p \rightarrow Y])) \ ,$$

and the term $X[.\mathrm{Id} \rightarrow Y]$ *as a shorthand form for* $X \stackrel{\circ}{=} Y$.

For a path functional dependency

$$c(p_1 \cdots p_k \rightarrow p_{k+1} \cdots p_n)$$

the path functional dependency formulae are

$$\begin{aligned} X[p \rightarrow P] \longleftarrow\ & X : c[p_1 \rightarrow P_1; \ldots ; p_k \rightarrow P_k] \wedge \\ & Y : c[p_1 \rightarrow P_1; \ldots ; p_k \rightarrow P_k; p \rightarrow P] \end{aligned}$$

for each $p \in \{p_{k+1}, \ldots , p_n\}$.

- *The set of path functional dependencies in* SC_D *is denoted by* PFD_D.

In the definition above, we defined the left and right sides of path functional dependencies to be sequences of path-functions. Despite this definition we use them often with sets of path-functions as left and right sides.

From now onwards, we limit the set of semantic constraints in a schema to class inclusion constraints, onto constraints and path functional dependencies, i. e., a schema consists only of the following components unless otherwise said

$$D = \langle \, \mathrm{ST}_D | \underbrace{\mathrm{CIC}_D \cup \mathrm{OC}_D \cup \mathrm{PFD}_D}_{\Xi_D :=} \, \rangle \quad .$$

We simply call a set of class inclusion constraints, onto constraints and path functional dependencies a set of *constraints* in the remainder of this work.

Example 29
We extend the schema in Exam. 13 by the following path functional dependencies

$$\{f(.n \to .o), f''(.k.m.q \to .k.l), f''(.k.l \to .n),\\ h(.q \to .p.r)\} \quad .$$

So whenever we refer to the schema S in Exam. 13, we understand from now on that these path functional dependencies are included.

Interesting in this set of path functional dependencies is the dependency

$$f''(.k.l \to .n) \quad ,$$

because the path-function $.n$ appears in its right-hand side, although the attribute n is not declared on the class f'' nor any of its superclasses. But the attribute n is "inherited" due to the class inclusion constraint $f'' \subset f$, which holds for every instance of the schema S.

Again as in the case for class inclusion constraints and onto constraints, we need inference rules for path functional dependencies to make the decision problem of the implication amenable to an algorithmic treatment.

To be more precise, we need inference rules for class inclusion constraints, onto constraints and path functional dependencies. It turns out that class inclusion constraints and onto constraints affect the implication of path functional dependencies, but not the other way round. The impact of class inclusion constraints and onto constraints already starts at the declaration of path functional dependencies, because these constraints potentially enlarge the set of path-functions for a class. Because of this one-way influence, a fact that we prove later in Theor. 67, the inference rules I1 to I5 are even complete when path functional dependencies are added to the set of semantic constraints. Then we supplement these inference rules to capture the implication of class inclusion constraints, onto constraints and path functional dependencies.

Weddell [Wed89, Wed92] presents five inference rules for path functional dependencies alone, which we adapt to our data model as well. The first rule (*I6: path functional dependency reflexivity*) captures the reflexivity of path functional dependencies. The second rule (*I7: path-function augmentation*) allows to extend both sides of a path functional dependency by a set of path-functions. The third rule (*I8: path functional

dependency transitivity) shows that path functional dependencies are transitive. These three rules exist in similar forms, with only attributes instead of path-functions, for relational functional dependencies.

The next two rules *simple attribution* and *simple prefix augmentation* have no counterparts in the relational data model. The rule *simple attribution* says that an object uniquely determines its attribute values. While the first four rules act locally on one class only, *simple prefix augmentation* includes two classes. If a path functional dependency can be derived for a class c_2 and this class is reachable from a class c_1 via the attribute a, then the path functional dependency that results from prolonging each path-function in the original path functional dependency declared for the class c_2 can be derived for the class c_1.

The following inference rule (*I11: simple prefix reduction*) is the inverse of simple prefix augmentation. Simple prefix reduction is described by Thalheim [Tha93] under the name *reduction*. Simple prefix reduction allows to telescope the path-functions in a derived path functional dependency provided the path-functions have a common prefix and an onto constraint is derivable with the common prefix as attribute. The last rule (*I12: path functional dependency inheritance*) brings class inclusion constraints into play. A path functional dependency declared for a class c_2 holds also for every subclass of the class c_2 where we use the term subclass to denote both real subclasses and those classes for which we can derive a corresponding class inclusion constraint.

For the inference rules I1 to I5 we use the notation $\vdash_D$ to denote a derivation of a class inclusion constraint or onto constraint. Since the inference rules I6 to I12 produce only path functional dependencies and therefore no conflicts arise, we reuse this notation even for these rules.

Definition 30 (Inference Rules for CICs, OCs and PFDs)
Let π be a path functional dependency over a schema D. The path functional dependency π is derivable from the set Ξ_D, written $\Xi_D \vdash_D \pi$, :iff it is a member of the set Ξ_D or the result of one or more applications of the following inference rules.

I6. Path functional dependency reflexivity: *For every class $c \in \mathrm{CL}_D$ and for all non-empty sets $Y \subset X$ of path-functions, where the set X is a finite subset of $\mathrm{PathFuncs}_D(c)$, derive $c(X \to Y)$.*

I7. Path-function augmentation: *For every class $c \in \mathrm{CL}_D$ and finite subset Z of $\mathrm{PathFuncs}_D(c)$, if $c(X \to Y)$ can be derived, then so can $c(XZ \to YZ)$ (where XZ, for example, denotes the union of all path-functions in X and Z).*

I8. Path functional dependency transitivity: *If both $c(X \to Y)$ and $c(Y \to Z)$ can be derived, then so can $c(X \to Z)$.*

I9. Simple attribution: *For every class $c \in \mathrm{CL}_D$ and attribute $a \in \mathrm{At}_D(c)$, derive $c(.\mathrm{Id} \to .a)$.*

I10. Simple prefix augmentation: *For every class $c_1 \in \mathrm{CL}_D$ and attribute $a \in \mathrm{At}_D(c_1)$, if*

$$c_2(p_1 \cdots p_k \to p_{k+1} \cdots p_n)$$

can be derived, where $c_2 \in \mathrm{Ran}_D(c_1, .a)$*, then so can*

$$c_1(.a \circ p_1 \cdots .a \circ p_k \rightarrow .a \circ p_{k+1} \cdots .a \circ p_n) \ .$$

I11. Simple prefix reduction: *For every class* $c_1 \in \mathrm{CL}_D$*, if*

$$c_2(.a \circ p_1 \cdots .a \circ p_k \rightarrow .a \circ p_{k+1} \cdots .a \circ p_n)$$

can be derived, where $c_2\{a|c_1\} \in \Upsilon_D^{+_D}$*, then so can*

$$c_1(p_1 \cdots p_k \rightarrow p_{k+1} \cdots p_n) \ .$$

I12. Path functional dependency inheritance: *For every class* $c_1 \in \mathrm{CL}_D$*, if* $c_2(X \rightarrow Y)$ *can be derived where* $c_1 \subset c_2 \in \Upsilon_D^{+_D}$*, then so can* $c_1(X \rightarrow Y)$*.*

Example 31

Figure 6 exhibits applications of the inference rules I6 to I12 using the semantic constraints in the schema S in Exam. 13. Remember that we included the path functional dependencies in Exam. 29 into the semantic constraints.

$$\begin{array}{rl}
 & \overset{\text{I6}}{\vdash}_S h(.q, .p \rightarrow .q) \\
h(.q \rightarrow .p.r) & \overset{\text{I7}}{\vdash}_S h(.q, .p \rightarrow .p.r, .p) \\
\left.\begin{array}{l} h(.q, .p \rightarrow .q) \\ h(.q \rightarrow .p.r) \end{array}\right\} & \overset{\text{I8}}{\vdash}_S h(.q, .p \rightarrow .p.r) \\
 & \overset{\text{I9}}{\vdash}_S h(.\mathrm{Id} \rightarrow .p) \\
h(.q \rightarrow .p.r) & \overset{\text{I10}}{\vdash}_S e(.m.q \rightarrow .m.p.r) \\
\left.\begin{array}{l} f''(.k.m.q \rightarrow .k.l) \\ f''\{k|e'\} \end{array}\right\} & \overset{\text{I11}}{\vdash}_S e'(.m.q \rightarrow .l) \\
\left.\begin{array}{l} f(.n \rightarrow .o) \\ f' \subset f \end{array}\right\} & \overset{\text{I12}}{\vdash}_S f'(.n \rightarrow .o)
\end{array}$$

Figure 6: Application of the inference rules I6 to I12

In the following sections we need to know which constraints, be it class inclusion constraints, onto constraints or path functional dependencies, we can derive from a set of constraints in a schema by means of the inference rules I1 to I12. Sometimes we are only interested in the path functional dependencies derivable from the set of constraints in a schema.

In the relational data model we determine the set of attributes over a relational scheme that are functionally determined by another set of attributes over this relational scheme. Likewise, we determine the set of path-functions starting at a class that is functionally determined by another set of path-functions starting at the same class.

Definition 32 (Closure for CICs, OCs and PFDs)

- *The* closure *of the set* Ξ_D*, written* $\Xi_D^{+_D}$*, is the set of all constraints* ξ *over the schema* D*, where* $\Xi_D \vdash_D \xi$*.*

- *The* PFD-closure *of the set* Ξ_D*, written* $\Xi_D^{\oplus_D}$*, is the set of all path functional dependencies* π *over the schema* D*, where* $\Xi_D \vdash_D \pi$*.*

- *The* closure of a finite, non-empty set of path-functions $X \subset \mathrm{PathFuncs}_D(c)$ *for some class* $c \in \mathrm{CL}_D$*, written* $X^{+_{D,c}}$*, is the set of all path-functions* $p \in \mathrm{PathFuncs}_D(c)$ *where* $\Xi_D \vdash_D c(X \to p)$*.*

The inference rules I1 to I12 are correct, i. e., every instance satisfying the semantic constraints satisfy every derivable constraint.

Theorem 33 (Soundness of the Inference Rules I1 to I12)
The inference rules I1 to I12 are sound*, i. e., for the set* Ξ_D *of constraints in a schema* D *and a constraint* ξ *over the schema* D*: if* $\xi \in \Xi_D^{+_D}$ *is a constraint, then every instance of the schema* D *satisfies the constraint* ξ*,* $\mathrm{sat}(D) \subset \mathrm{sat}(D \cup \{\xi\})$*.*

Proof. Showing the correctness of the inference rules is straightforward.

□

4 Completeness of the Inference Rules

Showing the completeness of the inference rules is definitely harder. The central idea is to show the completeness by contraposition. So instead of showing that when a constraint is satisfied by every instance of a schema, then this constraint is derivable from the constraints in the schema, we exhibit that when a constraint is not derivable from the constraints in a schema, there exists an instance of the schema that does not satisfy the constraint.

To facilitate this task, we decompose the completeness proof into two parts. First, we prove the completeness of the inference rules I1 to I5 and then the completeness of the inference rules I6 to I12. The independence of the inference rules I1 to I5 of path functional dependencies makes this decomposition feasible.

Before we embark on the completeness proofs, we point out that the inference rules I1 and I2 are complete for only class inclusion constraints. A fact that we exploit later on.

Lemma 34 (Completeness of the Inference Rules I1 to I2 for CICs)
Let $c \subset c'$ *be a class inclusion constraint over a schema* $D = \langle \mathrm{ST}_D | \Lambda \rangle$*, where* Λ *is a set of class inclusion constraints over the schema* D*. Then if the inclusion* $\mathrm{sat}(\langle \mathrm{ST}_D | \Lambda \rangle) \subset \mathrm{sat}(\langle \mathrm{ST}_D | \Lambda \cup \{c \subset c'\} \rangle)$ *holds, we can derive the class inclusion constraint* $c \subset c'$ *from the set* Λ *under the schema* D*,* $\Lambda \vdash_D c \subset c'$*.*

Proof. We conduct this proof by contraposition, i. e., we assume conversely that $\Lambda \not\vdash_D c \subset c'$ holds and show then that there exists an extension f with $f \in \mathrm{sat}(\langle \mathrm{ST}_D | \Lambda \rangle)$ and $f \notin \mathrm{sat}(\langle \mathrm{ST}_D | \Lambda \cup \{c \subset c'\} \rangle)$.

We construct the extension of the arbitrary schema D satisfying the axioms AX and the set Λ of class inclusion constraints but not the class inclusion constraint $c \subset c'$. This extension f consists of two objects o and o' where the object o is element of all classes d, for which we can derive $\Lambda \vdash_D c \subset d$, and the object o' is element of all classes. The object o' is the value of the object o for all attributes declared on all classes d, for which we can derive $\Lambda \vdash_D c \subset d$, and the value of the object o' for all attributes for all classes, because the object o' is element of all classes.

$$\begin{aligned} f := \langle\ & \{o{:}d \mid \Lambda \vdash_D c \subset d\} \cup \{o'{:}d \mid d \in \mathrm{CL}_D\} | \\ & \{o[a \to o'] \mid a \in \textstyle\bigcup_{d \in \{d' | \Lambda \vdash_D c \subset d'\}} \mathrm{At}_D(d)\} \cup \\ & \{o'[a \to o'] \mid a \in \textstyle\bigcup_{d \in \mathrm{CL}_D} \mathrm{At}_D(d)\}\ \rangle \end{aligned}$$

A sketch of the extension f is outlined in Fig. 7.

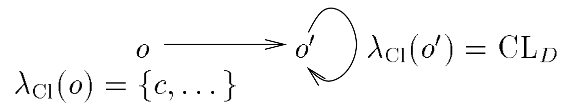


Figure 7: Sketch for the extension in the proof of Lem. 34

The extension f satisfies the unique-name axioms, not-null axiom and well-typedness axioms.

The objects o and o' satisfy the set of class inclusion constraints Λ by definition. Therefore the extension f is also an instance of the schema D.

Since the hypothesis $\Lambda \not\vdash_D c \subset c'$ holds, we know due to the inference rule I2 that $\mathrm{HR}_D \not\models c{::}c'$ holds, which means $\mathrm{compl}_D(f) \not\models c{::}c'$ holds by the proof theory of F-logic. But then the object o is not an element of the class c'. Therefore the extension f is not an instance of the schema $\mathrm{sat}(\langle \mathrm{ST}_D | \Lambda \cup \{c \subset c'\}\rangle)$.

□

4.1 Completeness of the Inference Rules I1 to I5

As already mentioned, we construct an extension f of a schema D such that the extension f is

- an instance of the schema D,
- but not an instance of the schema $D \cup \{\xi\}$,

where ξ is a constraint not derivable from the set of constraints in the schema D.

The idea is to start first with an extension that satisfies at least the set AX of axioms and later on to amend this extension to satisfy the semantic constraints in the schema D by upholding the invariant that the extension does not satisfy the constraint ξ. In the

case of the completeness proof for the inference rules I1 to I5, we call such a prototype instance an *S-tree* (Def. 41), and in the case of the inference rules I6 to I12 a *Two-S-graph* (Def. 58).

The way to construct such an S-tree for a schema D and a constraint v of the form $c \subset d$ or $d\{m|c\}$ is to take an object r of the class c in both cases as starting point. As mentioned before we want the constructed extension to satisfy first the set AX of axioms. Applying this insight to our extension, consisting so far only of the object r, means that the not-null axiom has to be satisfied. Thus we have to introduce appropriate values for the attributes of the object r. According to the well-typedness axioms, these attribute values are objects of some classes, which in turn have signatures declared on them. So again as for the object r, we have to introduce attribute values to satisfy the not-null axiom. This process where we always introduce new objects to avoid conflicts with the unique name axioms has to be iterated until all axioms are satisfied (Lem. 43). The outcome of this iteration has the form of a tree with the object r as *root*, hence the name *S-tree*. The letter S in this name stems from the fact that an S-tree represents an unfolding of the *s*chema graph starting at the class c. If the declaration of signatures in the schema D contains a cycle, e. g. declarations like $a[m \Rightarrow b]$ and $b[n \Rightarrow a]$, the tree becomes infinite. Therefore we allow infinite extensions and instances of a schema.

To keep track of the positions of objects in S-trees relative to the root, we adorn each object with a *path-label*, which is a path-function. A path-label indicates which attribute-value edges have to be traversed when we want to reach the corresponding object setting out at the root.

However, in general, an S-tree does not satisfy onto constraints. So as already said, we amend an S-tree in such a way that onto constraints are satisfied. When an onto constraint $d\{a|c\}$ is violated, there exists an object o, element of the class c, that is not referenced by an object of the class d via the attribute a. To remedy this violation, we insert a new object o' in the extension, make the object o' an element of the class d, and stipulate the object o as value for the attribute a of the object o'.

We soon discover we have to bestow some care making the insertion; we have to ensure that the set AX of axioms is satisfied again. So instead of inserting only one single object o', we insert an S-tree with the root o', which we prune by cutting off the branch starting with the attribute in the violated constraint, which is the attribute a in our example. We call such a pruned S-tree *pruned-S-tree* (Def. 44). We keep track of the inserted object again by means of labels, but this time we sometimes have to traverse attribute-value edges in the reverse direction and, therefore, employ way-descriptions as *way-labels*. The reverse traversal takes place exactly when we have to reach a root of a pruned-S-tree.

We simultaneously insert pruned-S-trees for all objects violating onto constraints and call this operation extrev (Def. 48), which upholds the invariant that if an extension satisfies the set AX of axioms, then extrev($\cdot$) satisfies the set AX of axioms (Lem. 50) as well.

Iterating this process of introducing new objects produces an extension that satisfies the onto constraints derivable from the constraints in the schema (Lem. 52).

In order to be in the position to prove that the final outcome is then a counterexample, the construction process is solely based upon the knowledge of what can be derived rather of what can be deduced.

To satisfy the class inclusion constraints derivable from the set of constraints in the schema, we follow a different strategy, which ensures that the derivable class inclusion constraints are satisfied from the outset. So when we begin with the root of the initial S-tree, we ensure that the root is an element of all classes required by the set of constraints but no other classes, and treat the class memberships of later inserted objects in the iteration process likewise. The iteration process does not alter the class membership of any object, so all objects satisfy all class inclusion constraints derivable from the set of constraints.

Finally, we know that the iteration process generates an output that satisfies both the set AX of axioms and the set of constraints in the schema but not the constraint υ, and thus we prove the completeness of the inference rules I1 to I5 (Theor. 53).

S-trees and the extensions we obtain by applying the operation extrev iteratively on S-trees are special extensions where each object carries a *way-label*, which indicates the objects' relative positions from the root(s). So each way-label stands in a special relationship with way-labels of the objects in its neighbourhood. Additionally, there is a relationship between the class-label of an object and its way-label. The class-label can be determined solely by knowing the class-label of the root and the way-label of the object. The two relationships are combined in the notion of an *extension with labelling.*

DEFINITION 35 (INSTANCE (EXTENSION) WITH LABELLING)
Let f be an instance (extension) of a schema $D = \langle \mathrm{ST}_D | \emptyset \rangle$, $C \subset \mathrm{CL}_D$ be a set of classes, and Υ be a set of class inclusion constraints and onto constraints over the schema D. The instance (extension) f is an instance (extension) with labelling based on the sets C and Υ*, if there is a* way-label $\lambda_{\mathrm{Wf}}(o) \in \bigcup_{c \in C} \mathrm{WayDes}_{D \cup \Upsilon}(c)$ *for each object* $o \in \mathrm{obj}(f)$ *with the following properties*

1. $\lambda_{\mathrm{Cl}}(o) = \bigcup_{c \in C \cap \mathrm{Dom}_{D \cup \Upsilon}(\lambda_{\mathrm{Wf}}(o))} \mathrm{Ran}_{D \cup \Upsilon}(c, \lambda_{\mathrm{Wf}}(o))$*, and*

2. *for all objects* $o' \in \mathrm{obj}(f)$*, if* $\mathrm{ob}_f \models o[a \to o']$*, then* $\lambda_{\mathrm{Wf}}(o) \circ .a = \lambda_{\mathrm{Wf}}(o')$*.*

We call an object $o \in \mathrm{obj}(f)$ *with* $\lambda_{\mathrm{Wf}}(o) = .\mathrm{Id}$ root *for the instance (extension) f. The set of roots for f,* $\{r | r \in \mathrm{obj}(f)$ *and* $\lambda_{\mathrm{Wf}}(r) = .\mathrm{Id}\}$*, is denoted* $\mathrm{Root}(f)$*.*

This definition is well defined due to Lem. 34, because the definition of a range of a way-description relies on the definition of a folder of a class, which in turn relies on the derivation of class inclusion constraints.

We choose way-labels to be way-descriptions over not only the schema D but also the schema $D \cup \Upsilon$; and thus lay the foundation for satisfying the axioms and derivable class inclusion constraints, because we do not have to change the initial class memberships in later steps. Property 1 of extensions with labellings ensures that the class memberships are correctly chosen, and property 2 ensures that way-labels are not arbitrarily chosen.

EXAMPLE 36
An instance with labelling for the schema $\langle \mathrm{ST}_S|\emptyset\rangle$*, the set* $\{h\}$ *of classes, and the set* Υ_S *of class inclusion constraints and onto constraints is sketched in Fig. 8, where* S *is the schema in Exam. 13. The object* o_h *is the only root of this instance with labelling.*

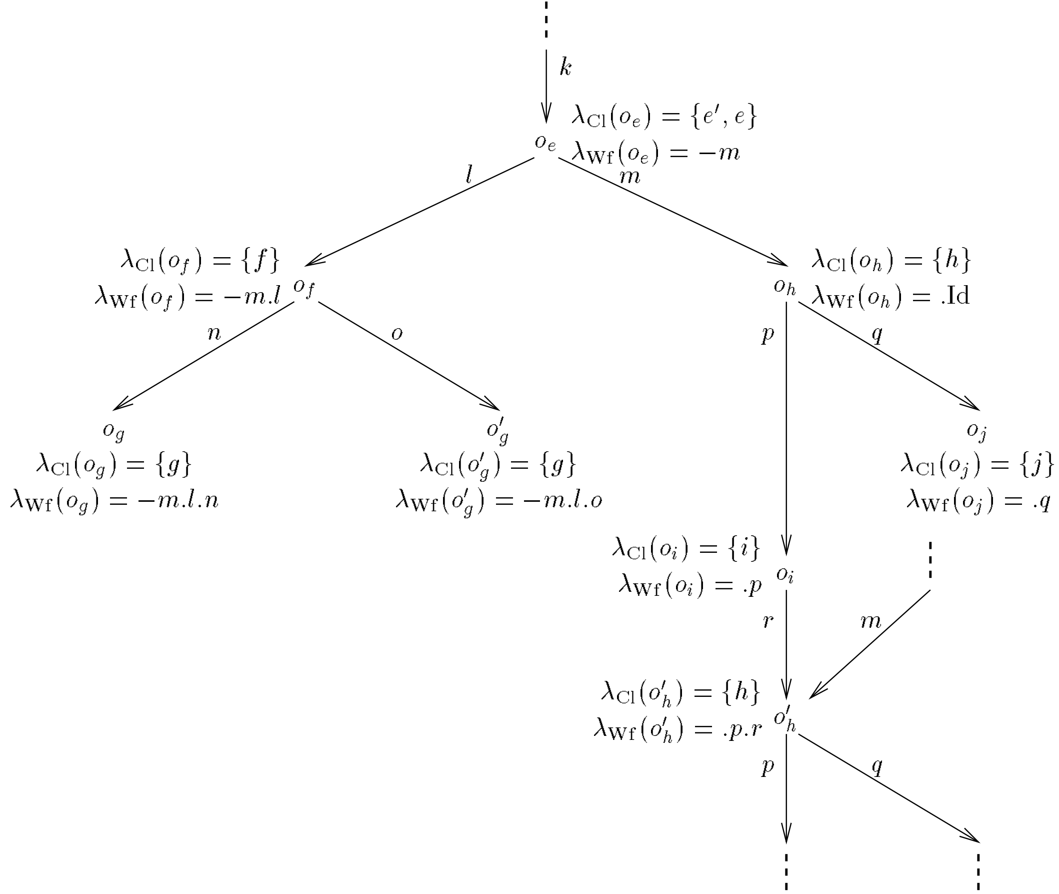


Figure 8: A sketch of the instance with labelling according to the parameters in Exam. 36

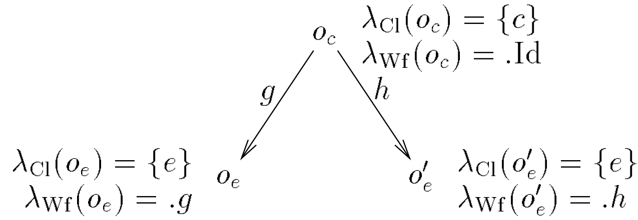
Unfortunately, by now we can only prove the inference rules I1 to I12 to be complete with restrictions imposed on the declaration of onto constraints.

Without any restrictions the operation extrev might transform an extension that satisfies path functional dependencies into one that violates path functional dependencies. For example, we consider a schema with the following components

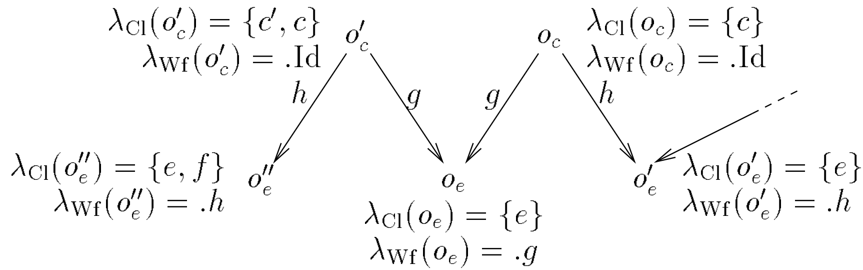
$$\begin{aligned} \mathrm{HR}_D &= \{c'::c\} \\ \mathrm{SG}_D &= \{c[g \Rightarrow e; h \Rightarrow e], c'[h \Rightarrow f]\} \ , \text{ and} \\ \mathrm{SC}_D &= \{c(.g \rightarrow .h), c'\{g|e\}\} \ . \end{aligned}$$

We look at the instance in Fig. 9(a). This instance satisfies the path functional dependency $c(.g \rightarrow .h)$. Alas, applying the operation extrev to this extension results in an

extension (Fig. 9(b)) that does not satisfy the path functional dependency $c(.g \rightarrow .h)$ because of the objects o_c and o'_c. These objects agree on their value for the attribute g but not on their value for the attribute h as required by the path functional dependency. However, the object o'_c has to be introduced to remedy the violation of the onto constraint $c'\{g|e\}$.



(a) An extension satisfying path functional dependencies



(b) A changed extension violating path functional dependencies

Figure 9: Reasons for the restriction on onto constraints

If we want to cure the violation of the path functional dependency, we might think of tampering with the class membership of the object o_c instead of introducing a new object, i. e., we might be tempted to make the object o_c a member of the class c'. But this is not a good idea, because then the resulting extension satisfies semantic constraints the original extension does not satisfy, and changing the class membership has a rippling effect on other objects as well. Because of the well-typedness axioms and the signature atom $c'[h \Rightarrow f]$, the object o_e must become a member of the class f.

A different solution might be to stipulate the object o'_e as value for the attribute h of the object o'_c. But again this has a rippling effect on the class memberships of objects. In this case the object o'_e has to become an object of the class f as before due to the well-typedness axioms and the signature atom $c'[h \Rightarrow f]$.

We take more drastic precautions by simply not allowing such situations in that we demand that whenever an onto constraint $d\{a|c\}$ over a schema is declared, this attribute a is a proper attribute for the class d.

Definition 37 (Proper Constraints)
Let Ξ_D be the set of constraints in a schema D. The set Ξ_D of constraints is a proper set of constraints *if for each onto constraint $d\{a|c\} \in \Xi_D$ the attribute a is a proper attribute for the class d.*

The next lemma shows that when we restrict the set of onto constraints in a schema to a proper set of constraints, situations as mentioned above, i. e. the violation of path functional dependencies, do not occur. This is because whenever an object o is referenced by an object via an attribute a, the reparation of violations of onto constraints never introduces another object referencing the object o via the attribute a, since this object o does not violate any onto constraint of the form $d\{a|c\}$ in the first place.

Lemma 38
Let f be an instance of a schema $D = \langle \mathrm{ST}_D | \emptyset \rangle$. Let Ξ be a set of proper constraints over the schema $D \cup \Xi$. Whenever $\mathrm{ob}_f \models o'[a \to o]$, $c \in \lambda_{\mathrm{Cl}}(o)$ and $c'\{a|c\} \in \Xi^{+_{D\cup\Xi}}$ holds, then $c' \in \lambda_{\mathrm{Cl}}(o')$.

Proof. The onto constraint $c'\{a|c\} \in \Xi^{+_{D\cup\Xi}}$ can only be derived by the inference rules I4 and I5 when an onto constraint $d'\{a|d\} \in \Xi$ exists such that $d' \subset c'$ and $c \subset d$ can be derived from the set Ξ of constraints. But then the attribute a is a proper attribute for the class d', $\mathrm{SG}_D \models d'[a \Rightarrow (\,)]$. Because of the well-typedness axioms there exists a signature atom $\mathrm{HR}_D \cup \mathrm{SG}_D \models e[a \Rightarrow (\,)]$ that covers $o'[a \to o]$, hence $e \in \lambda_{\mathrm{Cl}}(o')$, and therefore, because of the fact that the attribute a is a proper attribute for the class d', the class e is a subclass of the class d', $\mathrm{HR}_D \models e{::}d'$. Then we derive $e \subset d'$, and hence $e \subset c'$ by transitivity, which implies due to the correctness of the inference rules $c' \in \lambda_{\mathrm{Cl}}(o')$.

□

In the sequel we deal only with proper sets of constraints without saying this explicitly.

To show the completeness of the inference rules, we exploit a property of extensions with labelling, which is in a sense a generalisation of property 2 of extensions with labelling for attribute-value paths.

Lemma 39
Let f be an instance of a schema $D = \langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \mathrm{CL}_D$ of classes and a set Υ of class inclusion constraints and onto constraints over the schema D. Let $o \in \mathrm{obj}(f)$ be an object, $c \in \lambda_{\mathrm{Cl}}(o)$ be a class, and $p \in \mathrm{PathFuncs}_{D\cup\Upsilon}(c)$ be a path-function. Then there exists an object $o'' \in \mathrm{obj}(f)$ with $o.p = \{o''\}$ and $\lambda_{\mathrm{Wf}}(o) \circ p = \lambda_{\mathrm{Wf}}(o'')$.

Proof. We show this by induction on the length of the path-function p.

$\mathrm{len}(p) = 0$. Then $o.p = \{o\}$ according to Def. 25, and $\lambda_{\mathrm{Wf}}(o) \circ .\mathrm{Id} = \lambda_{\mathrm{Wf}}(o)$.

$\mathrm{len}(p) > 0$. Then the path-function p is of the form $p = p' \circ .a$, and $o.p' = \{o'\}$ according to the inductive assumption. Since f is an instance with labelling based on the sets C and Υ,

$$c \in \lambda_{\mathrm{Cl}}(o) = \bigcup_{c' \in C \cap \mathrm{Dom}_{D\cup\Upsilon}(\lambda_{\mathrm{Wf}}(o))} \mathrm{Ran}_{D\cup\Upsilon}(c', \lambda_{\mathrm{Wf}}(o)) \ ,$$

and hence there exists a class $c' \in C \cap \mathrm{Dom}_{D\cup\Upsilon}(\lambda_{\mathrm{Wf}}(o))$ with $c \in \mathrm{Ran}_{D\cup\Upsilon}(c', \lambda_{\mathrm{Wf}}(o))$. Since $p \in \mathrm{PathFuncs}_{D\cup\Upsilon}(c)$ is a path-function, there exist classes i and g such that $i \in \mathrm{Ran}_{D\cup\Upsilon}(c, p')$ and $\mathrm{HR}_{D\cup\Upsilon} \cup \mathrm{SG}_{D\cup\Upsilon} \models i[a \Rightarrow g]$. So $c' \in C \cap \mathrm{Dom}_{D\cup\Upsilon}(\lambda_{\mathrm{Wf}}(o) \circ p')$ with $i \in \mathrm{Ran}_{D\cup\Upsilon}(c', \lambda_{\mathrm{Wf}}(o) \circ p')$ by Lem. 24, and consequently due to the inductive assumption ($\lambda_{\mathrm{Wf}}(o) \circ p' = \lambda_{\mathrm{Wf}}(o')$) $c' \in C \cap \mathrm{Dom}_{D\cup\Upsilon}(\lambda_{\mathrm{Wf}}(o'))$ with $i \in \mathrm{Ran}_{D\cup\Upsilon}(c', \lambda_{\mathrm{Wf}}(o'))$, which means $i \in \lambda_{\mathrm{Cl}}(o')$, because f is an instance with labelling based on the sets C and Υ. Since $\mathrm{HR}_{D\cup\Upsilon} \cup \mathrm{SG}_{D\cup\Upsilon} \models i[a \Rightarrow g]$ holds, $\mathrm{HR}_D \cup \mathrm{SG}_D \models i[a \Rightarrow g]$ holds as well. This signature atom demands due to the unique name axioms and to the not-null axiom the existence of exactly one object o'' with

$$\mathrm{ob}_f \models o'[a \rightarrow o''] \ , \tag{1}$$

which leads to the conclusion $o.p = \{o''\}$. Additionally, (1) implies $\lambda_{\mathrm{Wf}}(o') \circ .a = \lambda_{\mathrm{Wf}}(o'')$ by property 2 of way-labellings. Finally, we have $\lambda_{\mathrm{Wf}}(o) \circ p = \lambda_{\mathrm{Wf}}(o) \circ (p' \circ .a) = (\lambda_{\mathrm{Wf}}(o) \circ p') \circ .a = \lambda_{\mathrm{Wf}}(o') \circ .a = \lambda_{\mathrm{Wf}}(o'')$.

□

An immediate consequence of this lemma is presented in the next corollary.

Corollary 40

Let f be an instance of a schema $D = \langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \mathrm{CL}_D$ of classes and a set Υ of class inclusion constraints and onto constraints over the schema D. For all objects $o, o' \in \mathrm{obj}(f)$, and for all classes $c \in \lambda_{\mathrm{Cl}}(o) \cap \lambda_{\mathrm{Cl}}(o')$ if $o.p = o'.p$ for $p \in \mathrm{PathFuncs}_{D\cup\Upsilon}(c)$, then $\lambda_{\mathrm{Wf}}(o) = \lambda_{\mathrm{Wf}}(o')$.

The prototype extension, an S-tree, for the counterexample of the completeness proof is an extension with labelling that contains only path-functions as way-labels, because an S-tree merely aims at satisfying the axioms and the derivable class inclusion constraints. An S-tree is very similar to a *C-Tree* defined by Weddell [Wed89, Wed92], where C stands in this case for a single class instead of a set of classes. Both kinds of trees possess adorned objects; but each C-Tree is an S-tree with labelling based on $\{C\}$ and the empty set of constraints.

The construction idea of an S-tree is to start with an object r, the *root* of the S-tree, to make the root an element of all classes derivable from a set C of classes and the set Υ of constraints, and to introduce new objects as attribute values of the root r due to the not-null axiom. These attribute values are made elements of classes according to the well-typedness axioms and the set Υ of constraints. We iterate this process of introducing attribute values for newly inserted objects. Instead of actually performing this process, we use path-functions, precisely those that start at the classes in the set C.

To keep the information of why we insert an object, we adorn the object with the corresponding path-function that leads to the insertion of this object. This adornment helps later on to prove properties of this object.

Definition 41 (S-Tree)
An S-tree *is an extension* $f := \langle \mathrm{pp}_f | \mathrm{ob}_f \rangle$ *of a schema* $D = \langle \mathrm{ST}_D | \emptyset \rangle$ *with labelling based on a set* $C \subset \mathrm{CL}_D$ *of classes and a set* Υ *of class inclusion constraints and onto constraints over the schema* D *constructed as follows.*

Step 1: *For each* $p \in \bigcup_{c \in C} \mathrm{PathFuncs}_{D \cup \Upsilon}(c)$ *take a new constant* v *and for all* $d \in \bigcup_{c \in C \cap \mathrm{Dom}_{D \cup \Upsilon}(p)} \mathrm{Ran}_{D \cup \Upsilon}(c, p)$ *add* $v : d$ *to* pp_f*, and add an additional label* $\lambda_{\mathrm{Wf}}(v)$ *assigned* p*.*

Step 2: *For each* $u, v \in \mathrm{obj}(f)$*, where* $p = \lambda_{\mathrm{Wf}}(u)$ *and* $p \circ .a = \lambda_{\mathrm{Wf}}(v)$*, add* $u[a \to v]$ *to* ob_f*.*

Example 42
An S-tree f *for the schema* $\langle \mathrm{ST}_S | \emptyset \rangle$*, the set* $\{h\}$ *of classes, and the set* Υ_S *of class inclusion constraints and onto constraints is outlined in Fig. 10, where* S *is again the schema in Exam. 13.*

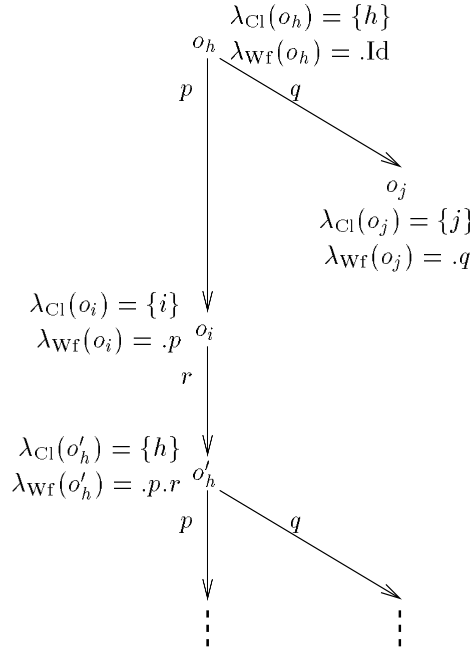


Figure 10: An outline of the S-tree f according to Exam. 42

To show that an extension f as defined above is an instance of the corresponding schema, we have to check that the completion of f under D satisfies the axioms. As we always take a new constant for objects and only one per path-function, a violation of unique name axioms is not an issue here. The addition of attribute values is based on path-functions as well. So no not-null and well-typedness axioms are violated. The satisfaction of the set Υ is not an issue at this point. The next lemma proves all as stated above.

Lemma 43 (S-Tree Instance)
Let f be an S-tree of a schema $D = \langle \mathrm{ST}_D | \emptyset \rangle$ based on a set $C \subset \mathrm{CL}_D$ of classes and a set Υ of class inclusion constraints and onto constraints over the schema D. Then the S-tree f is an instance of the schema D with labelling based on the sets C and Υ.

Proof. To show that the S-tree f is an instance of the schema D, we show that all axioms are satisfied. But before the presentation of this proof, we observe that for each object $o \in \mathrm{obj}(f)$, the set of classes this object o is element of can be syntactically determined due Def. 35 and Lem. 34,

$$\lambda_{\mathrm{Cl}}(o) = \bigcup_{c \in C \cap \mathrm{Dom}_{D \cup \Upsilon}(\lambda_{\mathrm{Wf}}(o))} \mathrm{Ran}_{D \cup \Upsilon}(c, \lambda_{\mathrm{Wf}}(o)) \ . \tag{2}$$

UN: The sound and complete proof theory of F-logic reveals that there are three possible ways to introduce equality between F-logic objects:

1. by explicitly stating the equality,
2. by is-a acyclicity, and
3. by scalarity.

The first possibility can be ruled out because of the definitions of schemas and extensions, which simply lack this possibility. The second possibility cannot occur, because the schema D has an acyclic class hierarchy. The third way cannot occur, because for each path-function only one object is introduced.

NN: Let $o \in \mathrm{obj}(f)$ be an object according to the definition of the S-tree f. This means for every attribute $a \in \mathrm{At}_{D \cup \Upsilon}(c')$ for some class $c' \in \lambda_{\mathrm{Cl}}(o)$ exists a class $c \in C \cap \mathrm{Dom}_{D \cup \Upsilon}(\lambda_{\mathrm{Wf}}(o))$ such that $c' \in \mathrm{Ran}_{D \cup \Upsilon}(c, \lambda_{\mathrm{Wf}}(o))$. Therefore $\lambda_{\mathrm{Wf}}(o) \circ .a \in \mathrm{PathFuncs}_{D \cup \Upsilon}(c)$ since $\lambda_{\mathrm{Wf}}(o)$ is a path-function. According to the definition of the S-tree f there exists then an object o' with $\lambda_{\mathrm{Wf}}(o') = \lambda_{\mathrm{Wf}}(o).a$, and hence $\mathrm{ob}_f \models o[a \rightarrow o']$.

WT: Let $\mathrm{ob}_f \models o[a \rightarrow o']$. Then $\lambda_{\mathrm{Wf}}(o) \circ .a = \lambda_{\mathrm{Wf}}(o')$.

1. Because the way-labels $\lambda_{\mathrm{Wf}}(o)$ and $\lambda_{\mathrm{Wf}}(o')$ are path-functions, there exists a class $c' \in \mathrm{Ran}_{D \cup \Upsilon}(c, \lambda_{\mathrm{Wf}}(o))$ for some class $c \in C \cap \mathrm{Dom}_{D \cup \Upsilon}(\lambda_{\mathrm{Wf}}(o))$ with $\mathrm{HR}_{D \cup \Upsilon} \cup \mathrm{SG}_{D \cup \Upsilon} \models c'[a \Rightarrow (\,)]$, hence $\mathrm{HR}_D \cup \mathrm{SG}_D \models c'[a \Rightarrow (\,)]$. Because of (2) the class c' is an element of $\lambda_{\mathrm{Cl}}(o)$, $c' \in \lambda_{\mathrm{Cl}}(o)$, and therefore $c'[a \Rightarrow (\,)]$ covers $o[a \rightarrow o']$.
2. Let $c \in \lambda_{\mathrm{Cl}}(o)$ be a class such that $\mathrm{HR}_D \cup \mathrm{SG}_D \models c[a \Rightarrow c']$ for some class c'. Due to (2), there exists $c'' \in C \cap \mathrm{Dom}_{D \cup \Upsilon}(\lambda_{\mathrm{Wf}}(o))$ such that $c \in \mathrm{Ran}_{D \cup \Upsilon}(c'', \lambda_{\mathrm{Wf}}(o))$. Finally, this means $c' \in \mathrm{Ran}_{D \cup \Upsilon}(c'', \lambda_{\mathrm{Wf}}(o) \circ .a)$, and hence $c' \in \lambda_{\mathrm{Cl}}(o')$.

□

In general an S-tree does not satisfy onto constraints. So we modify an S-tree by adding a new object whenever an onto constraint is violated. This object refers to the object

that gave rise to the violation via the corresponding attribute. But the addition of one object for every violation is not sufficient. The not-null axiom calls for the addition of an S-tree for every object violating onto constraints. Then such an S-tree is pruned. The branch that starts with the attribute of the violated onto constraint is removed, and the object that led to the violation is grafted instead. The way-labels of the objects of the pruned-S-tree are prolonged to adjust the relative position of the objects to the new context.

Definition 44 (Pruned-S-Tree)
Let $D = \langle \mathrm{ST}_D | \emptyset \rangle$ *be a schema,* $C \subset \mathrm{CL}_D$ *be a set of classes,* Υ *be a set of class inclusion constraints and onto constraints over the schema* D*,* $a \in \bigcup_{c \in C} \mathrm{At}_D(c)$ *be an attribute,* $w \in \mathrm{W_{wf}}(D \cup \Upsilon)$ *be a way-description such that* $w \circ -a$ *is a way-description, and* $o \in \mathcal{F}$ *be a new*[3] *constant. A* pruned-S-tree *is an extension* $\mathrm{pst}(C, o, a, w) := \langle \mathrm{pp} | \mathrm{ob} \rangle$ *of the schema* D *with labelling based on the sets* C *and* Υ *constructed as follows.*

Step 1: *For each* $p \in \bigcup_{c \in C} \mathrm{PathFuncs}_{D \cup \Upsilon}(c) \backslash \{x \in \mathrm{W_{wf}}(D \cup \Upsilon) \mid ex.\ y \in \mathrm{W_{arb}}(D \cup \Upsilon) \colon x \equiv .ay\}$ *take a new*[4] *constant* v *and for all* $d \in \bigcup_{c \in C \cap \mathrm{Dom}_{D \cup \Upsilon}(p)} \mathrm{Ran}_{D \cup \Upsilon}(c, p)$ *add* $v : d$ *to* pp*, and add an additional label* $\lambda_{\mathrm{Wf}}(v)$ *assigned* $w \circ -a \circ p$*.*

Step 2: *For each* $u, v \in \mathrm{obj}(\mathrm{pst}(C, o, a, w))$*, where* $\lambda_{\mathrm{Wf}}(u) \circ .m = \lambda_{\mathrm{Wf}}(v)$*, add* $u[m \to v]$ *to* ob*.*

Step 3: *Add* $o'[a \to o]$ *to* ob*, where* o' *is the object with way labelling* $w \circ -a$*,* $\lambda_{\mathrm{Wf}}(o') = w \circ -a$*, i. e.,* o' *is the object added for the path-function* .Id*. Because of this "historical" background we call the object* o' *a* root *as well.*

Example 45
When we look at the S-tree in Exam. 42, which was constructed for the schema $\langle \mathrm{ST}_S | \emptyset \rangle$*, the set* $C = \{h\}$ *of classes, and the set* Υ_S *of class inclusion constraints and onto constraints, we notice that the object* o_h *violates the onto constraint* $e'\{m|h\}$*. We fix this violation by inserting the object* o_e *and by making the object* o_h *the value for the attribute* m *of the object* o_e*. The object* o_e *is a member of the classes* e' *and* e*. So the not-null axiom requires that a value for the attribute* l *of this object is defined, because the attribute* l *is declared on the class* e'*. Therefore we insert not only the object* o_e *but instead the pruned-S-tree* $\mathrm{pst}(\{e', e\}, o_h, m, .\mathrm{Id})$ *shown in Fig. 11. The parameter* .Id *passed for the construction indicates where the root* o_e *of the pruned-S-tree is inserted relatively to the root of the S-tree, while the argument* o_h *indicates the absolute position of the pruned-S-tree.*

We determine the set of objects that violate onto constraints. Each of these objects should then be referred to by a root of a pruned-S-tree. It might be the case that an object violates several onto constraints having the same attribute. In this case only one pruned-S-tree is introduced.

[3] *New* means that the constant does not occur in $\mathrm{CL}_D \cup \mathrm{AT}_D$.

[4] Here, *new* means that the constant neither occurs in $\mathrm{CL}_D \cup \mathrm{AT}_D$ nor is element of the extension under construction.

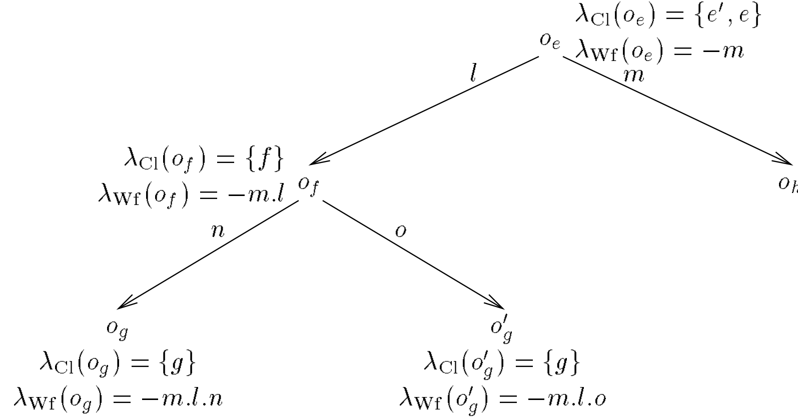


Figure 11: The pruned-S-tree $\mathrm{pst}(\{e', e\}, o_h, m, .\mathrm{Id})$ in Exam. 45

Definition 46 (Unsat)
Let f be an instance of a schema $D = \langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \mathrm{CL}_D$ of classes and a set Υ of class inclusion constraints and onto constraints over the schema D. The set of unsatisfied objects under Υ with the violated classes and attributes *is*

$$\mathrm{unsat}(f, \Upsilon) := \{(\{d \mid \textit{ex. } c\colon c \in \lambda_{\mathrm{Cl}}(o) \textit{ and } d\{a|c\} \in \Upsilon^{+_{D \cup \Upsilon}} \textit{ and } \textit{for all } o' \in \mathrm{obj}(f)\colon d \notin \lambda_{\mathrm{Cl}}(o') \textit{ or } \mathrm{ob}_f \not\models o'[a \to o]\}, o, a) \mid o \in \mathrm{obj}(f) \textit{ and } a \in \textstyle\bigcup_{c \in \mathrm{CL}_D} \mathrm{At}_D(c)\} \ .$$

Example 47
The set $\mathrm{unsat}(f, \Upsilon_S)$ for the S-tree f in Exam. 42 and the set Υ_S of class inclusion constraints and onto constraints in the schema S in Exam. 13 is

$$\mathrm{unsat}(f, \Upsilon_S) = \{(\{e', e\}, o_h, m), (\emptyset, o_h, n), \dots\} \ .$$

In Exam. 45 we utilised the information of the vector $(\{e', e\}, o_h, m)$ for the parameters in the construction of the pruned-S-tree $\mathrm{pst}(\{e', e\}, o_h, m, .\mathrm{Id})$ where the way-description $.\mathrm{Id}$ is the way-label of the object o_h, $\lambda_{\mathrm{Wf}}(o_h) = .\mathrm{Id}$.

In the next construction step we want to ensure that violated onto constraints are satisfied. Therefore, for every object that violates an onto constraint, we introduce a new object referring to that particular object. But instead of creating just one object, we insert a pruned-S-tree that has as root the inserted object.

Definition 48 (Extrev)
Let f be an instance of a schema $D = \langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \mathrm{CL}_D$ of classes and a set Υ of class inclusion constraints and onto constraints over the schema D. Then the extension $\mathrm{extrev}_\Upsilon(f)$ of the schema D with labelling based on the sets C and Υ is defined as

$$\mathrm{extrev}_\Upsilon(f) := f \cup \bigcup_{\substack{(C, o, a) \in \mathrm{unsat}(f, \Upsilon) \textit{ and} \\ C \neq \emptyset}} \mathrm{pst}(C, o, a, \lambda_{\mathrm{Wf}}(o)) \ ,$$

where every object introduced in one of the pruned-S-trees does not occur in neither any other pruned-S-tree nor the instance f.

Both the definition of unsat and the definition of extrev take instances of the underlying schema with labelling as input, because only the initial extension is an S-tree. The output of the operation extrev is then further used as input again. To enable this iteration, the output has to be an instance with labelling again as we show in Lem. 50.

EXAMPLE 49

Coming back to the S-tree f in Exam. 42, we apply the operation extrev *on this S-tree with the set Υ_S of class inclusion constraints and onto constraints in the schema S in Exam. 13. Since the vector $(\{e', e\}, o_h, m)$ is an element of the set* $\text{unsat}(f, \Upsilon_S)$*, we add among others the pruned-S-tree* $\text{pst}(\{e', e\}, o_h, m, \lambda_{\text{Wf}}(o_h))$ *as depicted in Fig. 11 and get the extension with labelling as shown in Fig. 12 as result.*

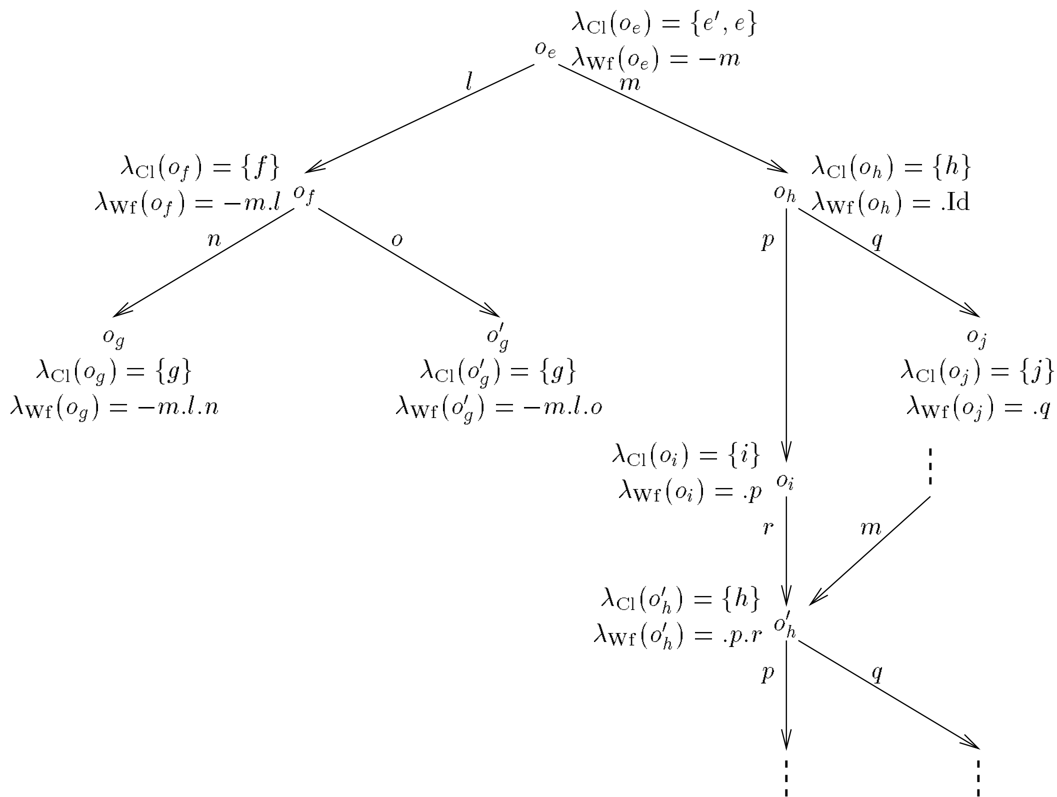


Figure 12: Operation extrev applied on the S-tree in Fig. 10

LEMMA 50 (EXTREV PRODUCES INSTANCES)

Let f be an instance of a schema $D = \langle \text{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \text{CL}_D$ of classes and a set Υ of class inclusion constraints and onto constraints over the schema D. Then $\text{extrev}_\Upsilon(f)$ *is an instance of the schema D with labelling based on the sets C and Υ.*

PROOF. That every pruned-S-tree is nearly an instance can be proven along the same lines of arguments as in the proof of Lem. 43 except for the satisfaction of the not-null axiom and well-typedness axioms for the pruned attribute. But then the not-null axiom is satisfied, because the pruned attribute receives the object violating onto constraints as value. The satisfaction of the well-typedness axioms for the pruned attribute remains to be shown.

So we consider the root o' of a pruned-S-tree introduced for the set of classes E, object o and attribute m, and $a \in \lambda_{\mathrm{Cl}}(o')$ be a class with

$$\mathrm{HR}_D \cup \mathrm{SG}_D \models a[m \Rightarrow b] \ . \tag{3}$$

According to the definition of a pruned-S-tree and Lem. 34

$$\lambda_{\mathrm{Cl}}(o') \overset{\mathrm{Def.35}}{=} \bigcup_{e \in E \cap \mathrm{Dom}_{D \cup \Upsilon}(.\mathrm{Id})} \mathrm{Ran}_{D \cup \Upsilon}(e, .\mathrm{Id}) \overset{\mathrm{Def.22}}{=} \bigcup_{e \in E} \mathrm{Folder}_{D \cup \Upsilon}(e) \ .$$

Therefore, there exists a class $a' \in E$ with $a' \subset a$, $a'\{m|b'\} \in \Upsilon^{+_{D \cup \Upsilon}}$ and $b' \in \lambda_{\mathrm{Cl}}(o)$. By inference rule domain relaxation, we have

$$a\{m|b'\} \in \Upsilon^{+_{D \cup \Upsilon}} \ . \tag{4}$$

Thus $b' \subset b$ follows from signature inclusion with (3) and (4). Consequently, we have $b \in \lambda_{\mathrm{Cl}}(o)$ and hence the satisfaction of all well-typedness axioms.

Conditions 1 and 2 of Def. 35 are also satisfied, because every pruned-S-tree is nearly an instance with labelling except for the root of the pruned-S-tree and the pruned attribute. But the root of the pruned-S-tree satisfies condition 1 by definition as well as condition 2. □

The next corollary draws the connection between Lem. 38 and the operation extrev.

COROLLARY 51
Let f be an instance of a schema $D = \langle \mathrm{ST}_D | \emptyset \rangle$. Let Ξ be a set of proper constraints over the schema $D \cup \Xi$. If $(C, o, m) \in \mathrm{unsat}(f, \Xi)$ and $\mathrm{ob}_f \models o'[m \to o]$, then the set C is empty, $C = \emptyset$.

Having the instance with labelling that violates the onto constraints in a set of constraints, we can fix the violations by applying the operation extrev to that instance; but we purchase thereby new violations of onto constraints, which we fix by iterating the application of the operation. The final outcome is then an instance of the underlying schema D with labelling that satisfies the set of constraints as well.

LEMMA 52
Let D be a schema, and t be an S-tree of the schema $\langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \mathrm{CL}_D$ of classes and the set Υ_D of class inclusion constraints and onto constraints in the schema D. Then $\bigcup_{i \in \mathbb{N}} \mathrm{extrev}^i_\Upsilon(t)$ is an instance of the schema D with labelling based on the sets C and Υ_D.

PROOF. Because of Lem. 50, $\text{extrev}^i_\Upsilon(t)$ is an instance of the schema $\langle \text{ST}_D|\emptyset\rangle$ with labelling based on the sets C and Υ_D provided t is an instance of the schema $\langle \text{ST}_D|\emptyset\rangle$ with labelling based on the sets C and Υ_D. Therefore $\bigcup_{i\in\mathbb{N}} \text{extrev}^i_\Upsilon(t)$ is an instance of the schema $\langle \text{ST}_D|\emptyset\rangle$ with labelling based on the sets C and Υ_D. It remains to show that the extension $\bigcup_{i\in\mathbb{N}} \text{extrev}^i_\Upsilon(t)$ satisfies the set SC_D of constraints.

Due to Def. 35 and Lem. 34 each derivable class inclusion constraint is satisfied in any extension $\text{extrev}^i_\Upsilon(t)$ and hence in $\bigcup_{i\in\mathbb{N}} \text{extrev}^i_\Upsilon(t)$.

If object o is generated in the i-th iteration, then all onto constraints that are violated by o are satisfied in the $i+1$-th iteration. This continues to be so throughout further iterations.

Let $c \in \text{CL}_D$ be a class and $o, o' \in \text{obj}(\bigcup_{i\in\mathbb{N}} \text{extrev}^i_\Upsilon(t))$ be two objects such that $c \in \lambda_{\text{Cl}}(o)$ and $c \in \lambda_{\text{Cl}}(o')$, and $o.p = o'.p$ for some path-function $p \in \text{PathFuncs}_D(c)$. According to Cor. 40 it follows that $\lambda_{\text{Wf}}(o) = \lambda_{\text{Wf}}(o')$. The generation of the S-tree t produces only one object for each way-label, and the operation extrev_Υ retains this invariant due to Cor. 51, hence $o = o'$. Therefore each path functional dependency in the set SC_D is trivially satisfied.

□

Finally we are ready to prove the completeness of the inference rules I1 to I5 for class inclusion constraints, onto constraints and path functional dependencies. The proof will be conducted by contraposition. We assume that we cannot derive a constraint v from the set Ξ_D of constraints, $\Xi_D \not\vdash_D v$. Then we construct an instance f of the schema D such that the extension f is

- an instance of the schema D,
- but not an instance of the schema $D \cup \{v\}$.

THEOREM 53 (COMPLETENESS OF THE INFERENCE RULES I1 TO I5)
Let D be a schema, and v be a class inclusion constraint or onto constraint over the schema D such that v cannot be derived from the set Ξ_D of constraints in the schema D, $\Xi_D \not\vdash_D v$. Then there exists an instance f of the schema D such that f is not an instance of the schema $D \cup \{v\}$.

PROOF. Let the constraint v be of the form $c \subset d$ or $d\{m|c\}$. Then let t be an S-tree built for $\langle \text{ST}_D|\emptyset\rangle$, the sets $\{c\}$ and Υ_D, and the object r be its root, $\text{Root}(t) = \{r\}$. Due to Lem. 43, the S-tree t is an instance of the schema $\langle \text{ST}_D|\emptyset\rangle$ with labelling, and therefore, by Lem. 52, $f := \bigcup_{i\in\mathbb{N}} \text{extrev}^i_\Upsilon(t)$ is an instance of the schema D with labelling based on the sets $\{c\}$ and Υ_D. It remains to show that the extension f does not satisfy the constraint v.

We prove by contradiction that the extension f does not satisfy the constraint v, be it a class inclusion constraint or an onto constraint. So we assume conversely that the extension f satisfies the constraint v.

If the constraint υ takes on the form $c \subset d$, then the root r is a member of the class d, $d \in \lambda_{\text{Cl}}(r)$. Due to Def. 35, the fact that the S-tree is constructed only for the class c and the fact that the object r is a root, $\lambda_{\text{Wf}}(r) = .\text{Id}$, the equation

$$\lambda_{\text{Cl}}(r) = \text{Folder}_D(c) \tag{5}$$

holds. So in contradiction to the assumption we derive $c \subset d$, because $d \in \lambda_{\text{Cl}}(r) = \text{Folder}_D(c)$.

If the constraint υ takes on the form $d\{m|c\}$, then there exists an object o with $\text{ob}_f \models o[m \to r]$. Because of the construction of the S-tree t, the S-tree t does not satisfy the onto constraint $d\{m|c\}$. So the object o must have been introduced in the first application of the operation extrev on the S-tree t. This means there exists a vector $(C, r, m) \in \text{unsat}(t, \Upsilon_D)$ with $C \neq \emptyset$. Consequently, there is a class $c' \in \lambda_{\text{Cl}}(r)$ and an onto constraint $d'\{m|c'\} \in \Upsilon_D^{+_D}$ such that $d \in \text{Folder}_D(d')$, hence $d' \subset d$ and $d\{m|c'\} \in \Upsilon_D^{+_D}$, because the onto constraint can only be satisfied if m is an attribute declared for the class d, $m \in \text{At}_D(d)$. Finally, (5) holds in this case as well and (5) implies $c \subset c'$, and therefore we derive the onto constraint $d\{m|c\}$ from the set Υ_D, $d\{m|c\} \in \Upsilon_D^{+_D}$, which is a contradiction.

□

4.2 Completeness of the Inference Rules I6 to I12

Knowing that the inference rules I1 to I5 are complete under class inclusion constraints, onto constraints and path functional dependencies, we give an outline of how we prove the completeness of the inference rules I6 to I12 under these constraints. The basic idea is to show the completeness by contraposition. So instead of showing for a path functional dependency $c(X \to Y)$ that if every instance of a schema D is an instance of the schema $D \cup \{c(X \to Y)\}$, then the path functional dependency is derivable from the set Ξ_D, we construct an instance of the schema D not satisfying the path functional dependency $c(X \to Y)$, if the dependency $c(X \to Y)$ is not derivable from the set Ξ_D.

In principle, the construction of the counterexample exploits two ideas. The core of the counterexample generalises the well-known Armstrong construction for similar proofs in the relational data model, using two tuples agreeing exactly on the closure of a set of attributes X. In our case we use two objects of the class c. But, in general, two objects are not sufficient, so we take two S-trees with their roots being members of the class c. Then these roots have to agree exactly on their values for path-functions in the closure $X^{+_{D,c}}$, because we want the roots to violate the path functional dependency $c(X \to Y)$. Thus we merge the two S-trees by removing some nodes, exactly those whose way-labels appear in the closure $X^{+_{D,c}}$, from one of the S-trees and afterwards by inserting missing attribute values for objects in the S-tree whose nodes got removed. These attribute values are objects of the S-tree that is still intact and carry the corresponding way-labels as the removed objects. The result, called *Two-S-graph*, satisfies the path functional dependencies in the schema D, but not the path functional dependency $c(X \to Y)$ (Lem. 62) and thus forms the prototype of the counterexample.

A Two-S-graph looks like a "Siamese twin": the roots are the heads and the objects with way-labels in the closure $X^{+_{D,c}}$ are the limbs that are grown together.

The satisfaction of class inclusion constraints is reached by carefully choosing the class memberships of the objects in the Two-S-graph.

Again we face the problem that a Two-S-graph does not satisfy onto constraints and simply applying the operation extrev on a Two-S-graph does not suffice, because, in general, the outcome does not satisfy the derivable path functional dependencies. For that we have to merge objects being inserted in different pruned-S-trees. Looking only at the newly inserted objects suffices, because once two objects satisfy the path functional dependencies, the objects reachable from these objects by path-functions remain unchanged (Lem. 65). We call the operation that first performs the operation extrev and then merges objects, merge (Def. 63).

As with the operation extrev, applying the operation merge leads to new violations of onto constraints, thus we iterate the process over and over again. The final outcome is then the counterexample we are looking for: It is an instance of the schema D but does not satisfy the path functional dependency $c(X \to Y)$.

The outcome of the iteration has still some similarity with the initial Two-S-graph, which looks like a "Siamese twin". First of all it is possible to reach every object by following the object's way-label starting from one of the roots. We call this property *root-reachability* (Def. 54). Secondly, there are only two roots, and, thirdly, if we reach one object by applying the object's way-label to one of the roots, there exists another object reachable from the other root by applying the first object's way-label to this other root. This property is called *root-isomorphism* (Def. 57). These two properties are exploited later on.

We begin with the definition of root-reachability, which is a property of instances with labelling being invariant under the transformation merge.

Definition 54 (Root-Reachability)
Let D be a schema, and f be an instance of the schema $\langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \mathrm{CL}_D$ of classes and the set Υ_D. The instance f is root-reachable, *if for all objects $o \in \mathrm{obj}(f)$ there exists a root $r \in \mathrm{Root}(f)$ such that $r.\lambda_{\mathrm{Wf}}(o) = \{o\}$.*

A root-reachable instance has the property that whenever we apply an appropriate way-description to a root, we get at most one object as result. But before we show this property, we prove whenever an object in an instance with labelling is reachable from a root by an appropriate way-description, this very way-description is the object's way-label.

Lemma 55
Let f be an instance of a schema $D = \langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \mathrm{CL}_D$ of classes and a set Υ of class inclusion constraints and onto constraints over the schema D. We have that for all objects $o \in \mathrm{obj}(f)$, for all roots $r \in \mathrm{Root}(f)$, for all classes $c \in \lambda_{\mathrm{Cl}}(r)$, and for all way-descriptions $w \in \mathrm{WayDes}_{D \cup \Upsilon}(c)$, if $o \in r.w$, then $\lambda_{\mathrm{Wf}}(o) = w$.

Proof. We show this by induction on the length of way-description w.

$\mathrm{len}(w) = 0$. This means that $w = .\mathrm{Id}$ and therefore $\lambda_{\mathrm{Wf}}(r) = .\mathrm{Id}$.

$\mathrm{len}(w) > 0$. This means w is of the form $w' \circ \pi m$ where $\pi \in \{., -\}$ and $m \in \mathrm{AT}_D$. Let $o \in r.w$. We distinguish two cases according to the structure of π:

$\pi = .$, then there exists $o' \in r.w'$ with $\mathrm{ob}_f \models o'[m \to o]$ and $\lambda_{\mathrm{Wf}}(o') = w'$ by the inductive hypothesis. Additionally, we know that w' does not end in $-m$, because w is a well-formed way-description and is of the form $w = w'.m$. Then $\lambda_{\mathrm{Wf}}(o) = \lambda_{\mathrm{Wf}}(o') \circ .m = w' \circ .m = w'.m = w$ because of Def. 35.

$\pi = -$, then there exists $o' \in r.w'$ with $\mathrm{ob}_f \models o[m \to o']$ and $\lambda_{\mathrm{Wf}}(o') = w'$ by the inductive hypothesis. Additionally, we know that w' does not end in $.m$, because w is a well-formed way-description and is of the form $w = w'{-m}$. Now we assume two cases:

$\lambda_{\mathrm{Wf}}(o) = y{-m}$, then $y = y \circ (-m \circ .m) = (y \circ -m) \circ .m = y{-m} \circ .m = \lambda_{\mathrm{Wf}}(o) \circ .m \overset{\text{Def. 35}}{=} \lambda_{\mathrm{Wf}}(o') = w'$ and therefore $\lambda_{\mathrm{Wf}}(o) = w'{-m} = w$.

$\lambda_{\mathrm{Wf}}(o) = y\pi'm'$, $\pi' \in \{., -\}$, $m' \in \mathrm{AT}_D$, $\pi'm' \neq -m$, then we know the following $w' = \lambda_{\mathrm{Wf}}(o') \overset{\text{Def. 35}}{=} \lambda_{\mathrm{Wf}}(o) \circ .m = y\pi'm' \circ .m = y\pi'm'.m$, which is a contradiction, because w' does not end in $.m$.

□

LEMMA 56

Let D be a schema, and f be a root-reachable instance of the schema $\langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \mathrm{CL}_D$ of classes and the set Υ_D. Then for all roots $r \in \mathrm{Root}(f)$, for all classes $c \in \lambda_{\mathrm{Cl}}(r)$, and for all way-descriptions $w \in \mathrm{WayDes}_D(c)$: $|r.w| \leq 1$.

PROOF. We show this by induction on the length of way-description w.

$\mathrm{len}(w) = 0$. Trivial.

$\mathrm{len}(w) > 0$. So w is of the form $w' \circ \pi m$ where $\pi \in \{., -\}$ and $m \in \mathrm{AT}_D$, and $|r.w'| \leq 1$ by the induction hypothesis. If $|r.w'| = 0$, then we know that $|r.w| = 0$.

So the case that $|r.w'| = 1$, say $r.w' = \{o\}$, needs to be investigated. We consider two cases according to the structure of π.

$\pi = .$, then $|r.w| = |o..m| = 1$.

$\pi = -$, then $\lambda_{\mathrm{Wf}}(o) \overset{\text{Lem.55}}{=} w'$. We assume now $o_1, o_2 \in r.w$ with $o_1 \neq o_2$. Again we know $\lambda_{\mathrm{Wf}}(o_1) = \lambda_{\mathrm{Wf}}(o_2) = w$ because of Lem. 55, and $\mathrm{ob}_f \models o_1[m \to o] \wedge o_2[m \to o]$. Then there exists a root $r' \in \mathrm{Root}(f)$ such that $r'.w = \{o_1\}$ because of the root-reachability of the instance f. Since $\{o_1, o_2\} \subset r.w$ and $o_1 \neq o_2$, r and r' are different objects, $r \neq r'$. Now $o \notin r'.w'$, because if we assume that $o \in r'.w'$ as in Fig. 13(a), then $\{o_1, o_2\} \in r'.w$, which is a contradiction. Then there exists o_1' with $o_1' \in r'.w'$ as in Fig. 13(b), and therefore $\mathrm{ob}_f \models o_1[m \to o_1'] \wedge o_1[m \to o]$, which is a contradiction to the scalarity of m. This means $|r.w| \leq 1$.

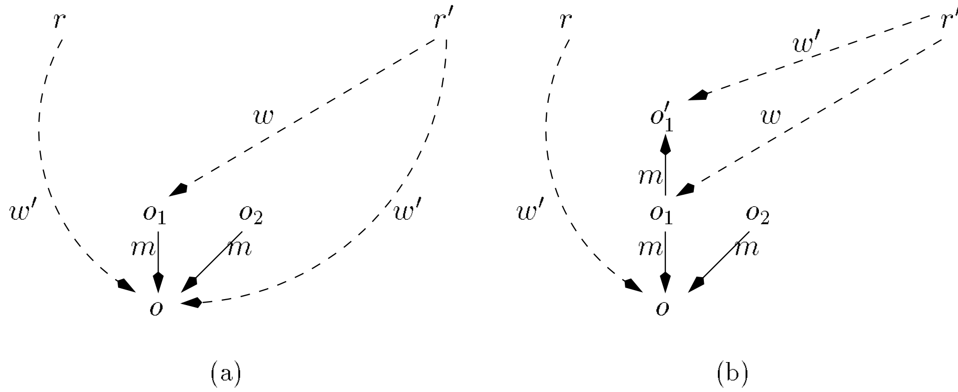


Figure 13: Situations that cannot arise in the case $\text{len}(w) > 0$ and $\pi = -$

□

In the sequel, we will write $r.w = o$ for a root r in a root-reachable instance instead of $r.w = \{o\}$.

As mentioned previously, the extensions obtained during the construction of the counterexample look the same whichever root we assume as vista point. This feature is captured by the property root-isomorphism. Again this property is invariant under the transformation merge.

Definition 57 (Root-Isomorphism)
Let D be a schema, and f be a root-reachable instance of the schema $\langle \text{ST}_D|\emptyset\rangle$ with labelling based on a set $C \subset \text{CL}_D$ of classes and the set Υ_D. The instance f is root-isomorphic *if*

- *the instance f has at most two roots, $|\text{Root}(f)| \leq 2$, and*
- *for all roots $r \in \text{Root}(f)$, for all objects $o \in \text{obj}(f)$ with $r.\lambda_{\text{Wf}}(o) = \{o\}$, and for all roots $r' \in \text{Root}(f)$ there exists an object $o' \in \text{obj}(f)$ such that $r'.\lambda_{\text{Wf}}(o) = \{o'\}$.*

The prototype for the counterexample is a Two-S-graph, which we compose out of two S-trees, hence the name Two-S-graph. From one of these trees we remove objects and create links to the other S-tree by inserting objects of the other S-tree for missing attribute values.

Definition 58 (Two-S-Graph)
A Two-S-graph *for a schema D and a path functional dependency $c(X \to Y)$ over the schema D is an extension $f = \langle \text{pp}_f|\text{ob}_f\rangle$ of the schema $\langle \text{ST}_D|\emptyset\rangle$ with labelling based on the sets $\{c\}$ and Υ_D constructed as follows.*

Step 1: *Construct two S-trees $f_1 := \langle \text{pp}_1|\text{ob}_1\rangle$ and $f_2 := \langle \text{pp}_2|\text{ob}_2\rangle$ of the schema $\langle \text{ST}_D|\emptyset\rangle$ with labelling based on the sets $\{c\}$ and Υ_D such that $\text{obj}(f_1)\cap\text{obj}(f_2) = \emptyset$. Let R_1, R_2 denote the single element in $\text{Root}(f_1)$, $\text{Root}(f_2)$, respectively.*

Step 2: *Remove any* $v : c_v \in \mathrm{pp}_2$ *and* $u[m \to v] \in \mathrm{ob}_2$ *whenever* $\lambda_{\mathrm{Wf}}(v) \in X^{+_{D,c}}$.

Step 3: $\mathrm{pp}_f := \mathrm{pp}_1 \cup \mathrm{pp}_2$ *and* $\mathrm{ob}_f := \mathrm{ob}_1 \cup \mathrm{ob}_2$.

Step 4: *For each* $u : c_u \in \mathrm{pp}_f$ *and* $m \in \mathrm{At}_D(c_u)$ *where due to Step 2* $\mathrm{ob}_f \not\models u[m \to w]$ *for all* $w \in \mathrm{obj}(f)$*, add* $u[m \to R_1.\lambda_{\mathrm{Wf}}(u).m]$ *to* ob_f.

EXAMPLE 59
The Two-S-graph (Fig. 14(b)) built for the schema in Exam. 13 and the path functional dependency $h(.q \to .p)$ *is composed out of two S-trees (Fig. 14(a)). The closure of the set* $\{.q\}$ *contains the path-function* $.q$*, therefore we remove the object* x_j *and insert the object* o_j *as attribute value for the object* x_h*. Because of the path functional dependency* $h(.q \to .p.r)$ *the path-function* $.p.r$ *is an element of the closure* $\{.q\}^{+_{S,h}}$*, hence the removal of the object* x'_h *and all objects reachable from the object via path-functions.*

A Two-S-graph constructed for a schema satisfies at least the set AX of axioms.

LEMMA 60 (TWO-S-GRAPH INSTANCE)
Let g *be a Two-S-graph for a schema* D *and a path functional dependency* $c(X \to Y)$ *over the schema* D*. The Two-S-graph* g *is an instance of the schema* $\langle \mathrm{ST}_D | \emptyset \rangle$.

PROOF. The two S-trees needed for the construction of the Two-S-graph g are instances of the schema $\langle \mathrm{ST}_D | \emptyset \rangle$ according to Lem. 43. Replacing some objects of one S-tree by objects of the other S-tree does not lead to a violation of the unique-name axioms, not-null axiom or well-typedness axioms.

□

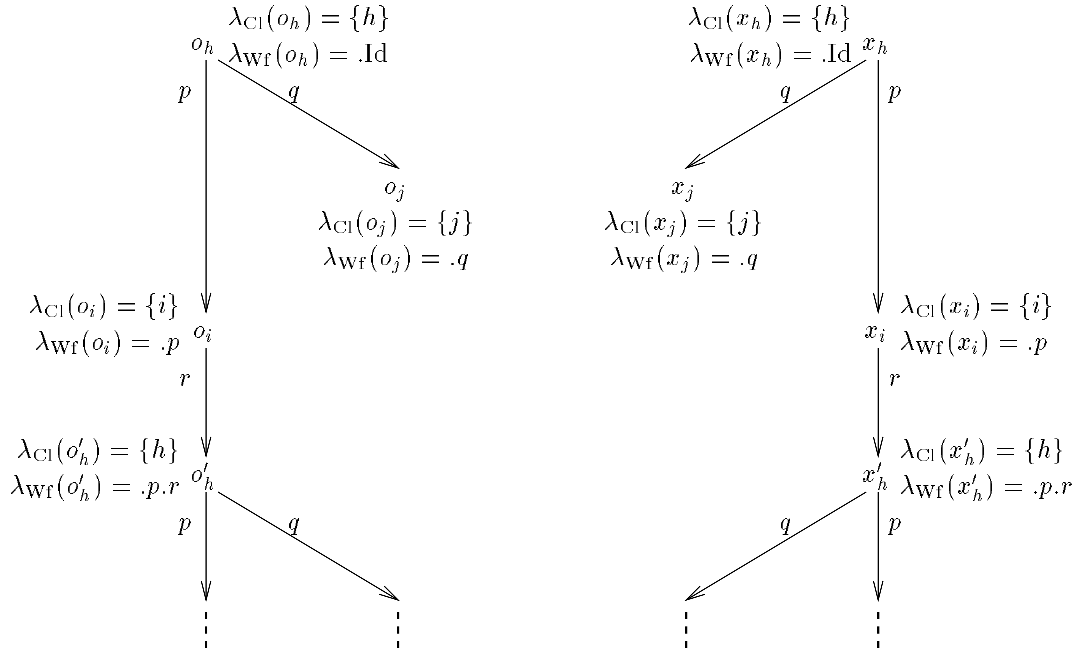
The construction of a Two-S-graph ensures that only those objects are removed whose way-labels are elements of the closure of the corresponding set of path-functions.

COROLLARY 61
Let g *be a Two-S-graph for a schema* D *and a path functional dependency* $c(X \to Y)$ *over the schema* D*. Then for all objects* $o \in \mathrm{obj}(g)$, $R_1.\lambda_{\mathrm{Wf}}(o) = R_2.\lambda_{\mathrm{Wf}}(o) = o$ *iff* $\lambda_{\mathrm{Wf}}(o) \in X^{+_{D,c}}$.

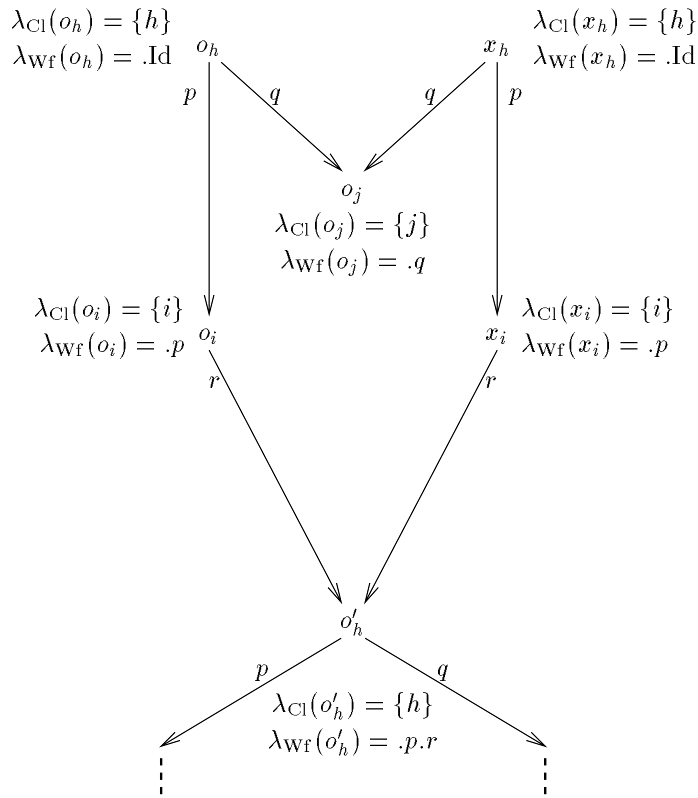
A Two-S-graph is indeed a prototype of the counterexample in that a Two-S-graph potentially satisfies the derivable path functional dependencies but not the path functional dependency the Two-S-graph is constructed for. Additionally, a Two-S-graph looks like a "Siamese twin", i. e., it is root-isomorphic.

LEMMA 62
Let D *be a schema,* $c(X \to Y) \notin \Xi_D^{+_D}$ *be a path functional dependency over the schema* D *not derivable from the set* Ξ_D*. Let* g *be a Two-S-graph for the schema* D *and the path functional dependency* $c(X \to Y)$.

1. *The Two-S-graph* g *is root-isomorphic.*
2. (a) *The Two-S-graph* g *is an instance of the schema* $\langle \mathrm{ST}_D | \Xi_D^{\oplus_D} \rangle$.
   (b) *The Two-S-graph* g *is not an instance of the schema* $\langle \mathrm{ST}_D | \{c(X \to Y)\} \rangle$.



(a) The two S-trees for the Two-S-graph



(b) The Two-S-graph

Figure 14: Constructing the Two-S-graph in Exam. 59

PROOF FOR 1. The Two-S-graph is an instance of the schema $\langle \mathrm{ST}_D|\emptyset\rangle$ according to Lem. 60.

For each object o inserted in one of the S-trees f_1 or f_2 during the construction of the Two-S-graph g, we know that the way-label $\lambda_{\mathrm{Wf}}(o)$ is a path-function. Without loss of generality, we assume that o has been inserted into the S-tree f_1. Then there exists an object o' with $R_1.\lambda_{\mathrm{Wf}}(o) = \{o'\}$ and $\lambda_{\mathrm{Wf}}(o') \overset{\text{Lem.55}}{=} \lambda_{\mathrm{Wf}}(R_1) \circ \lambda_{\mathrm{Wf}}(o) = .\mathrm{Id} \circ \lambda_{\mathrm{Wf}}(o) = \lambda_{\mathrm{Wf}}(o)$; but only one object is introduced for each S-tree and path-function, and therefore $o = o'$. The removal of nodes and edges from one of the S-trees and the addition of missing links does not mar this property, and thus the Two-S-graph is root-reachable.

A similar argumentation holds for the property root-isomorphism. The introduction of objects in the S-trees f_1 and f_2 during the construction of the Two-S-graph g is executed according to the set of path-functions starting at the class c. So whenever an object is reachable from a root of the Two-S-graph g either this object is reachable by the other root as well or the object with the same way-label. Therefore the Two-S-graph g is root-isomorphic, because the Two-S-graph g contains only the roots R_1 and R_2 by definition.

PROOF FOR 2a. Let

$$c'(p_1 \cdots p_k \to p_{k+1} \cdots p_n) \in \Xi_D^{+_D} \tag{6}$$

be a path functional dependency, $o, o' \in \mathrm{obj}(g)$ be two objects such that $c' \in \lambda_{\mathrm{Cl}}(o) \cap \lambda_{\mathrm{Cl}}(o')$, and

$$o.p_i = o'.p_i \text{ for all } i \in \{1, \ldots, k\} \ . \tag{7}$$

Then $w := \lambda_{\mathrm{Wf}}(o) = \lambda_{\mathrm{Wf}}(o')$ because of Cor. 40. We consider two cases:

1. There exists a root $r \in \mathrm{Root}(g)$ such that $o, o' \in r.w$. Then $o = o'$ because g is root-reachable, and therefore $|r.w| \leq 1$ (Lem. 56).
2. There does not exist a root $r \in \mathrm{Root}(g)$ such that $o, o' \in r.w$. Then there are roots $r, r' \in \mathrm{Root}(g)$ with $r.w = o \neq o' = r'.w$. Because of $c' \in \lambda_{\mathrm{Cl}}(o)$ and g is an extension with labelling, $c' \in \mathrm{Ran}_D(c, w)$, and therefore we can derive

$$c(w \circ p_1 \cdots w \circ p_k \to w \circ p_{k+1} \cdots w \circ p_n) \tag{8}$$

from (6) by simple prefix augmentation. Because of (7) $r.w \circ p_i = r'.w \circ p_i$ for $i \in \{1, \ldots, k\}$. By Cor. 61, it follows that $w \circ p_i \in X^{+_{D,c}}$ for $i \in \{1, \ldots, k\}$. This means $w \circ p_j \in X^{+_{D,c}}$ for $j \in \{k+1, \ldots, n\}$ because of (8). Consequently, $r.w \circ p_j = r'.w \circ p_j$ for $j \in \{k+1, \ldots, n\}$, and hence $o.p_j = o'.p_j$ for $j \in \{k+1, \ldots, n\}$, which means (6) is satisfied.

PROOF FOR 2b. The Two-S-graph g does not satisfy $c(X \to Y)$ because $c(X \to Y) \notin \Xi_D^{+_D}$, and therefore there is $p \in Y \backslash X^{+_{D,c}}$, hence $R_1.p \neq R_2.p$ by Cor. 61.

□

Having the prototype of the counterexample, we deal with the satisfaction of onto constraints. The basic idea is to insert a pruned-S-tree whenever an object violates an onto constraint. Unfortunately, this insertion entails in general a violation of path functional dependencies, which we cure by removing objects from except from one all of the inserted pruned-S-trees and adding missing links.

Therefore we determine the set of objects that are reachable by more than one root (the set shared). By means of this set, we identify for each root of a pruned-S-tree the set of paths $\pi(r'')$ that lead to these objects. Paths in the "closure" of the set $\pi(r'')$ lead to objects that are to be removed unless the path is already element of the set $\pi(r'')$. The "closure" $\Pi(r'')$ of the set $\pi(r'')$ has to be calculated with diligence because, in general, the set $\pi(r'')$ is non-finite and its elements, which are path-functions, start at different classes.

Definition 63 (Merge)
Let D be a schema, and f be a root-isomorphic instance of the schema $\langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on a set $C \subset \mathrm{CL}_D$ of classes and the set Υ_D. The transformation $\mathrm{merge}_D(f)$ is defined as the outcome of the following steps.

Let $r \in \mathrm{Root}(f)$ be an arbitrary root. We define (Fig. 15)

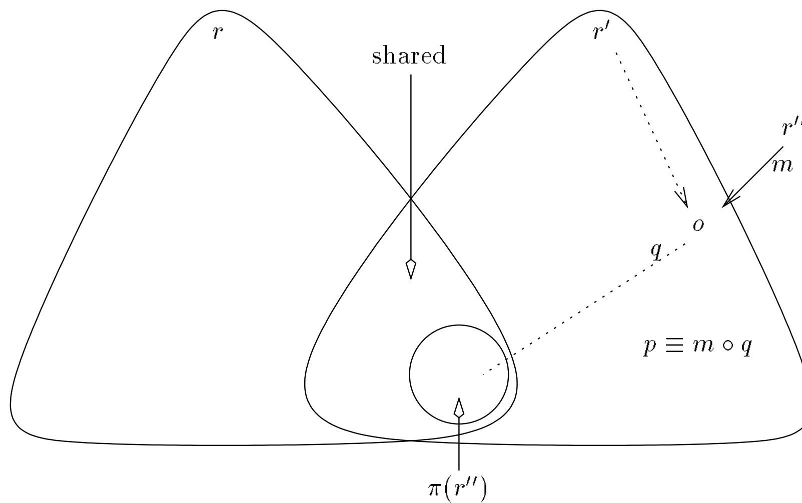


Figure 15: The sets shared and $\pi(r'')$ in Def. 63

$$\mathrm{shared} = \{u \in \mathrm{obj}(f) \mid \textit{ex. } r' \in \mathrm{Root}(f) \backslash \{r\}\colon\ r.\lambda_{\mathrm{Wf}}(u) = r'.\lambda_{\mathrm{Wf}}(u)\} \ .$$

For every object r'' that is the root of a pruned-S-tree $\mathrm{pst}(C', o, m, \lambda_{\mathrm{Wf}}(o))$ newly inserted in $\mathrm{extrev}_\Upsilon(f)$, and that is reachable from $r' \in \mathrm{Root}(f)\backslash\{r\}$ ($r'' \in r'.\lambda_{\mathrm{Wf}}(r'')$) and not reachable from r ($r'' \notin r.\lambda_{\mathrm{Wf}}(r'')$), we define

$$\pi(r'') = \{p \in \bigcup_{d \in \lambda_{\mathrm{Cl}}(r'')} \mathrm{PathFuncs}_D(d) \mid r''.p \subset \mathrm{shared}\} \ ,$$

and

$$\Pi(r'') = \bigcup_{d \in \lambda_{\mathrm{Cl}}(r'')} \left(\bigcup_{\emptyset \neq N \subset \pi(r'') \cap \mathrm{PathFuncs}_D(d) \ \textit{and}\ |N| \in \mathbb{N}} N^{+_{D,d}} \right) .$$

Then we apply the following steps to $f' := \mathrm{extrev}_\Upsilon(f)$.

1. *For each* $p \in \Pi(r'') \backslash \pi(r'')$ *remove the object* u *with* $r''.p = \{u\}$ *and any incident edge into* u *from* f'.
2. *For each remaining object* o'*, class* $d \in \lambda_{\mathrm{Cl}}(o')$*, and attribute* $m \in \mathrm{At}_D(d)$ *where due to the first step* $\mathrm{ob}_{f'} \not\models o'[m \to o'']$ *for all remaining objects* o'' *add* $o'[m \to r.\lambda_{\mathrm{Wf}}(o') \circ .m]$.

We designate extensions satisfying the derivable path functional dependencies as input for the operation merge. As the set X of path-functions was taken in the construction of a Two-S-graph to determine which objects had to be removed, the set $\pi(r'')$ for the root r'' of a newly inserted pruned-S-tree is used for this job as well. Applying a path-function from the set $\pi(r'')$ to the root r'' yields an object that is a member of the set shared, which comprises all objects reachable from all roots.

Example 64

The Two-S-graph in Exam. 59 violates the onto constraint $e'\{m|h\}$ *for two objects,* o_h *and* x_h*. So we insert for both of them a pruned-S-tree, but this extension (Fig. 16(a)) violates the path functional dependency* $e'(.m.q \to .l)$ *derivable from the set of constraints in the schema* S*. We remedy this violation by removing the object* x_f *and stipulating the object* o_f *as value for the attribute* l *of the object* x_e *(Fig. 16(b)).*

The removal of objects in the operation merge leaves the objects in the original instance untouched.

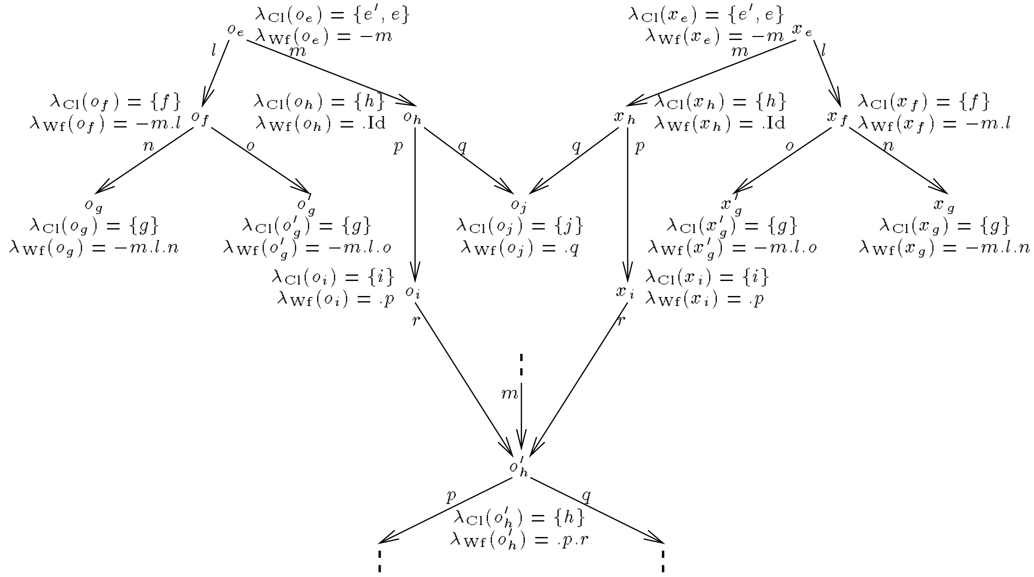
Lemma 65

Let D *be a schema, and* f *be a root-isomorphic instance of the schema* $\langle \mathrm{ST}_D | \emptyset \rangle$ *with labelling based on a set* $C \subset \mathrm{CL}_D$ *and the set* Υ_D*, and an instance of the schema* $\langle \mathrm{ST}_D | \Xi^{\oplus_D} \rangle$*. In the construction process of* $\mathrm{merge}_D(f)$ *no object* $\bar{o} \in \mathrm{obj}(f)$ *is removed, i. e. for all objects* $\bar{o} \in \mathrm{obj}(f)$*,* $\bar{o} \in \mathrm{obj}(\mathrm{merge}_D(f))$.

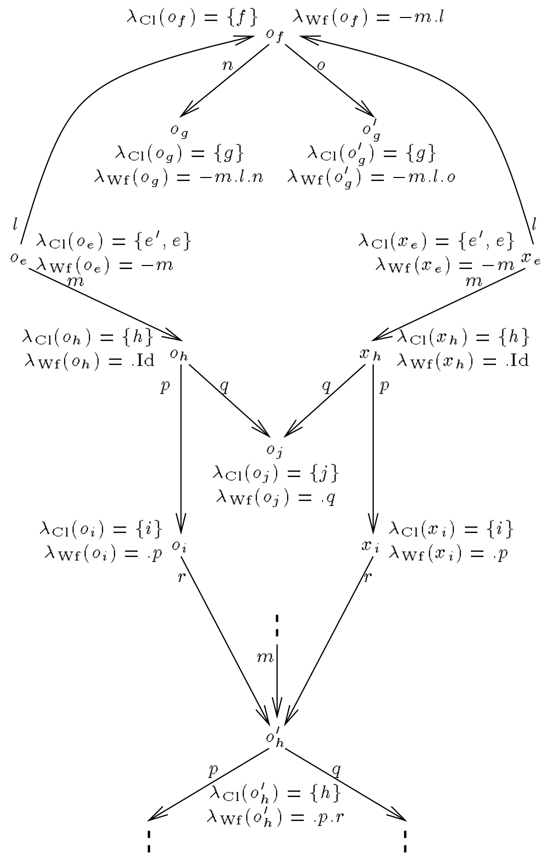
Proof. We show $\bigcup_{p \in \Pi(r'') \backslash \pi(r'')} r''.p \cap \mathrm{obj}(f) = \emptyset$ for the root r'' of any pruned-S-tree $\mathrm{pst}(C', o, m, \lambda_{\mathrm{Wf}}(o))$ newly inserted in $\mathrm{extrev}_\Upsilon(f)$. Let $r' \neq r$ be a root of the extension f' in Def. 63 with $r'.\lambda_{\mathrm{Wf}}(r'') = \{r''\}$.

Fig. 17 illustrates the resulting situation, which cannot arise as shown in the subsequent reasoning.

Let us conversely assume $v \in r''.p \cap \mathrm{obj}(f)$ for a $p \in \Pi(r'') \backslash \pi(r'')$. Then $v \notin$ shared by definition. Because of $r'.\lambda_{\mathrm{Wf}}(r'') \circ p = \{v\}$ and the root-isomorphism there exists $v' \in r.\lambda_{\mathrm{Wf}}(r'') \circ p \cap \mathrm{obj}(f)$ distinct from v, $v \neq v'$. Every path-function in $\pi(r'')$ has got the prefix $.m$ due to its definition. The path-function p has got the prefix $.m$ since $v \in \mathrm{obj}(f)$. $p \in \Pi(r'')$ and therefore $d(N \to p) \in \Xi_D^{+_D}$ for a class $d \in \lambda_{\mathrm{Cl}}(r'')$



(a) First applying the operation extrev, ...



(b) ..., then merging objects

Figure 16: Applying the operation merge on the Two-S-graph in Fig. 14(b)

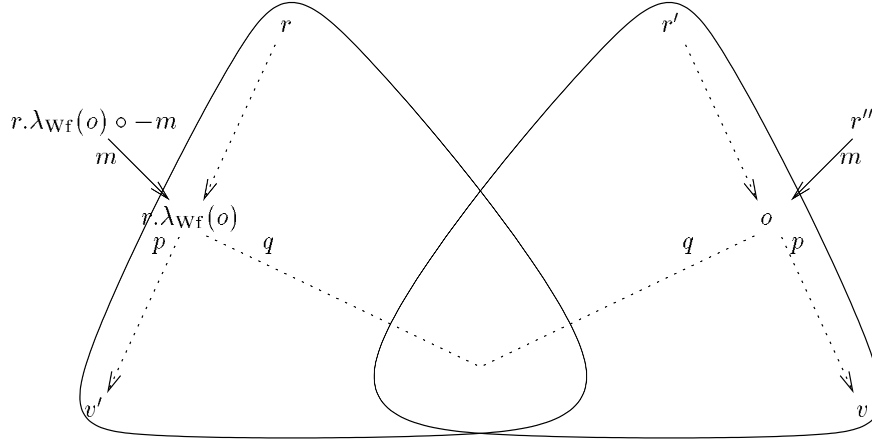


Figure 17: Objects in obj(f) are not touched by the operation merge

and some finite, non-empty set of path-functions $N \subset \pi(r'') \cap \mathrm{PathFuncs}_D(d)$. The root r'' was added because of $d''\{m|d'\} \in \Upsilon_D^{+_D}$ for some class d'' with $d'' \subset d$ and for some class $d' \in \lambda_{\mathrm{Cl}}(o)$. Then $d''(N \to p) \in \Xi_D^{+_D}$ holds by path functional dependency inheritance. Due to simple prefix reduction, this entails $d'(\alpha(.m, N) \to \alpha(.m, \{p\})) \in \Xi_D^{+_D}$ with $\alpha(P, S) := \{t \mid Pt \in S\}$. But then o and $r.\lambda_{\mathrm{Wf}}(o)$ violate this path functional dependency, because $o \neq r.\lambda_{\mathrm{Wf}}(o)$, $o.q = r.\lambda_{\mathrm{Wf}}(o) \circ q$ for each $q \in \alpha(.m, N)$ and $o.p' = v \neq v' = r.\lambda_{\mathrm{Wf}}(o) \circ p'$ for $\{p'\} = \alpha(.m, \{p\})$, which is a contradiction to the fact that f is an instance of $\langle \mathrm{ST}_D | \Xi^{\oplus_D} \rangle$.

□

A Two-S-graph forms the prototype for the counterexample, i.e., the Two-S-graph is a "Siamese twin", and satisfies the derivable path functional dependencies but not the path functional dependency it is constructed for. The operation merge treats these properties as invariants.

Lemma 66

Let D be a schema, $c(X \to Y) \notin \Xi_D^{+_D}$ be a path functional dependency over the schema D. Let f be a root-isomorphic instance of the schema $\langle \mathrm{ST}_D | \emptyset \rangle$ with labelling based on the sets $\{c\}$ and Υ_D, such that the instance f is an instance of the schema $\left\langle \mathrm{ST}_D | \Xi_D^{\oplus_D} \right\rangle$ and not an instance of the schema $\langle \mathrm{ST}_D | \{c(X \to Y)\} \rangle$.

1. *The extension* $\mathrm{merge}_D(f)$ *is root-isomorphic.*
2. *The extension* $\mathrm{merge}_D(f)$ *is an instance of* $\left\langle \mathrm{ST}_D | \Xi_D^{\oplus_D} \right\rangle$.
3. *The extension* $\mathrm{merge}_D(f)$ *is not an instance of* $\langle \mathrm{ST}_D | \{c(X \to Y)\} \rangle$.

Proof for 1. We show first that the extension $\mathrm{merge}_D(f)$ is root-reachable. Because of Lem. 65 and the root-reachability of the instance f, for each object $o \in \mathrm{obj}(f)$, there exists a root $r \in \mathrm{Root}(\mathrm{merge}_D(f))$ such that $r.\lambda_{\mathrm{Wf}}(o) = \{o\}$.

So we consider now objects added in the construction of $\text{merge}_D(f)$. Let $o \in \text{obj}(f)$ be an object for which a pruned-S-tree $\text{pst}(C', o, m, \lambda_{\text{Wf}}(o))$ has been introduced. Then the only access to an object in the pruned-S-tree is via the object o. But then there exists a root $r \in \text{Root}(\text{merge}_D(f))$ with $r.\lambda_{\text{Wf}}(o) = \{o\}$. The root o' of the pruned-S-tree has $\lambda_{\text{Wf}}(o) \circ -m$ as way-label and $\lambda_{\text{Wf}}(o)$ does not end in $.m$ due to Cor. 51. This implies $r.\lambda_{\text{Wf}}(o') = \{o'\}$, and therefore $r.\lambda_{\text{Wf}}(o'') = \{o''\}$ for each object o'' in the pruned-S-tree, because $\lambda_{\text{Wf}}(o'') = \lambda_{\text{Wf}}(o') \circ p$ for some path-function p. The removal of nodes and edges from pruned-S-trees and the addition of missing links does not impair this property, and thus the extension $\text{merge}_D(f)$ is root-reachable.

The same lines of arguments can be followed when proving the property root-isomorphism.

Proof for 2. The construction process for $\text{merge}_D(f)$ does not lead to a violation of AX. So we have to look at the path functional dependencies in $\Xi_D^{\oplus_D}$.

By contradiction. Let

$$c'(p_1 \cdots p_m \to p) \tag{9}$$

be a path functional dependency derivable from Ξ_D and $o, o' \in \text{obj}(\text{merge}_D(f))$ be two distinct objects violating this path functional dependency ($o.p_i = o'.p_i$ for $i \in \{1, \ldots, m\}$ implies $\lambda_{\text{Wf}}(o) = \lambda_{\text{Wf}}(o')$ by Cor. 40, and $o.p \neq o'.p$).

Because of the root-isomorphism and $\lambda_{\text{Wf}}(o) = \lambda_{\text{Wf}}(o')$, the objects o and o' are either both old objects, $o, o' \in \text{obj}(f)$, or newly inserted objects, $o, o' \in \text{obj}(\text{merge}_D(f)) \backslash \text{obj}(f)$. Since f is an instance of $\langle \text{ST}_D | \Xi_D^{\oplus_D} \rangle$ and because of Lem. 65, two objects in $\text{obj}(f)$ cannot violate (9), and so both objects must have been newly added in two distinct pruned-S-trees. Distinct because way-labels are unique in pruned-S-trees.

The situation described below is depicted in Fig. 18.

Let u, u' be the corresponding roots of the pruned-S-trees involved. Let $e'\{a|d''\}$ be the onto constraint that gave rise to the introduction of the root u.

The onto constraint $e'\{a|d''\}$ is derivable from a proper onto constraint $d'\{a|d''\}$ with $\Upsilon_D^{+_D} \vdash_D d' \subset e'$ and, by Lem. 38, gave rise to the introduction of the root u. Then the onto constraint $d'\{a|d''\}$ gave also rise to the introduction of the root u' because of $o.p_i = o'.p_i$ for $i \in \{1, \ldots, m\}$ and the root-isomorphism of the extension f.

Since all onto constraints are proper and $\lambda_{\text{Cl}}(u) = \text{Ran}_{\langle \text{ST}_D | \Upsilon_D \rangle}(c, \lambda_{\text{Wf}}(u))$, we have that $\Upsilon_D^{+_D} \vdash_D d' \subset e$ holds for all classes $e \in \lambda_{\text{Cl}}(u)$. Then there exists a unique $q \in \bigcup_{d \in \lambda_{\text{Cl}}(u)} \text{PathFuncs}_D(d)$ with $u.q = \{o\}$ and $u'.q = \{o'\}$. As a consequence u and u' violate

$$d'(q \circ p_1 \cdots q \circ p_m \to q \circ p) \tag{10}$$

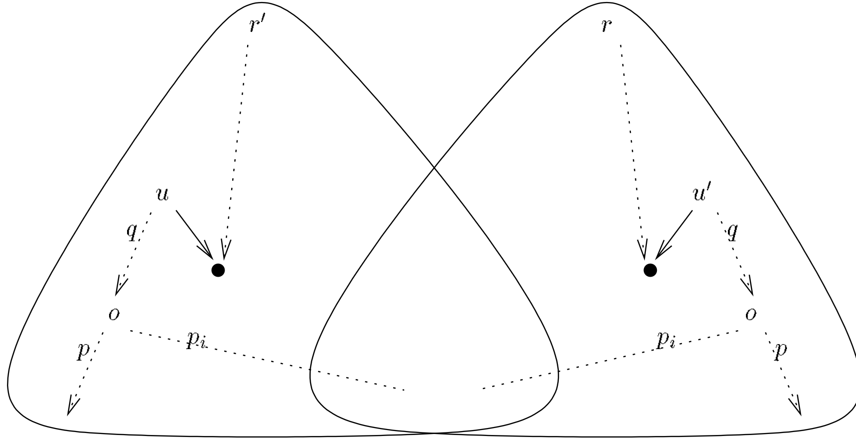


Figure 18: The situation that cannot arise in the proof for 2 of Lem. 66

for the class d', which can be derived from (9) by iterated application of simple prefix augmentation.

Since $u \neq u'$ and the root-reachability of $\text{merge}_D(f)$, there are distinct roots $r, r' \in \text{Root}(\text{merge}_D(f))$ such that $r.\lambda_{\text{Wf}}(u) = \{u\}$ and $r'.\lambda_{\text{Wf}}(u) = \{u'\}$. Without loss of generality let r be the root chosen in the construction of $\text{merge}_D(f)$. Then $\{q \circ p_1, \ldots, q \circ p_m\} \subset \Pi(u')$, because if $q \circ p_i \notin \Pi(u')$, then neither $u'.q \circ p_i \in \text{shared}$ nor $u'.q \circ p_i$ is removed in the first step of the construction of $\text{merge}_D(f)$ and therefore $o.p_i = u.q \circ p_i \neq u'.q \circ p_i = o'.p_i$, which contradicts $o.p_i = o'.p_i$.

Since there exist a finite, non-empty set of path-functions $N \subset \pi(u') \cap \text{PathFuncs}_D(d')$ such that $q \circ p_i \in N^{+_{D,d'}}$ and the path functional dependency (10), $q \circ p \in N^{+_{D,d'}}$ holds.

If we assume that

- $q \circ p \in \pi(u')$, then $u.q \circ p = u'.q \circ p$ by definition of $\pi(u)$, and
- $q \circ p \in N^{+_{D,d'}} \setminus \pi(u')$, then $u'.q \circ p$ is removed in the first step of the construction of $\text{merge}_D(f)$, and therefore by the second step of the construction $u.q \circ p = u'.q \circ p$.

The equality $u.q \circ p = u'.q \circ p$ contradicts that the path functional dependency (10) is violated. Therefore (9) cannot have been violated.

PROOF FOR 3. According to Lem. 65, the construction process does not touch the set $\text{obj}(f)$ of objects. Therefore the pair $o, o' \in \text{obj}(f)$ of objects that violates $c(X \to Y)$ in f still violates $c(X \to Y)$ in $\text{merge}_D(f)$.

□

Finally, we show the completeness of the inference rules I6 to I12 by contraposition. So we assume conversely that we cannot derive a path functional dependency $c(X \to Y)$

from the set of constraints in a schema D. Then we construct an instance of the schema D that does not satisfy the dependency $c(X \to Y)$.

THEOREM 67 (COMPLETENESS OF THE INFERENCE RULES I6 TO I12)
Let D be a schema, and $c(X \to Y) \notin \Xi_D^{+_D}$ be a path functional dependency over the schema D. Then there exists an instance f of the schema D that is not an instance of the schema $D \cup \{c(X \to Y)\}$.

PROOF. Let g be a Two-S-graph for the schema D and the path functional dependency $c(X \to Y)$. As proven in Lem. 62, the Two-S-graph g is a root-isomorphic instance of the schema $\langle \mathrm{ST}_D | \Xi_D^{\oplus_D} \rangle$ but not an instance of the schema $\langle \mathrm{ST}_D | \{c(X \to Y)\} \rangle$. From Lem. 66, we know that the operation merge_D leaves these properties untouched such that $f := \bigcup_{i \in \mathbb{N}} \mathrm{merge}_D^i(g)$ is an instance of the schema $\langle \mathrm{ST}_D | \Xi_D^{\oplus_D} \rangle$ but does not satisfy the path functional dependency $c(X \to Y)$.

Due to Def. 35 and Lem. 34 each derivable class inclusion constraint is satisfied in every extension $\mathrm{merge}_D^i(g)$ and hence f.

If object o is generated in the i-th iteration, then all onto constraints violated by o are satisfied in the $i+1$-th iteration. This continues throughout further iterations.

Therefore the extension f is an extension of the schema D that does not satisfy the path functional dependency $c(X \to Y)$.

□

5 Conclusion

In this work, we presented a axiomatisation for class inclusion constraints, onto constraints and path functional dependencies. We proved that the axiomatisation is sound and complete only under a minor restriction on onto constraints.

The axiomatisation lays the foundation for an algorithmic treatment of the implication of class inclusion constraints, onto constraints and path functional dependencies. The decidability of the implication is still an open problem for class inclusion constraints, onto constraints and path functional dependencies, but it is solved for path functional dependencies alone [IW94] in a semantic data model.

The results of this work can be put to use in computer aided, object-oriented database design [Pol00, BP00a, BP00b]. This is achieved by defining desirable properties of database schemas by means of class inclusion constraints, onto constraints and path functional dependencies and by defining database transformation that produce database schemas with the desirable properties.

References

[Bis95] Joachim Biskup. Database schema design theory: Achievements and challenges. In Subhash Bhalla, editor, *Proceedings of the 6th International Conference Information Systems and Management of Data*, number 1006 in Lecture Notes in Computer Science, pages 14–44, Bombay, 1995. Springer-Verlag.

[BMP96] Joachim Biskup, Ralf Menzel, and Torsten Polle. Transforming an entity-relationship schema into object-oriented database schemas. In J. Eder and L. A. Kalinichenko, editors, *Advances in Databases and Information Systems, Moscow 95*, Workshops in Computing, pages 109–136. Springer-Verlag, 1996.

[BMPS96] Joachim Biskup, Ralf Menzel, Torsten Polle, and Yehoshua Sagiv. Decomposition of relationships through pivoting. In Bernhard Thalheim, editor, *Proceedings of the 15th International Conference on Conceptual Modeling*, number 1157 in Lecture Notes in Computer Science, pages 28–41, Cottbus, Germany, 1996.

[BP00a] Joachim Biskup and Torsten Polle. Decomposition of database classes under path functional dependencies and onto constraints. In B. Thalheim K.-D. Schewe, editor, *Proceedings of the Foundations of Information and Knowledge Base Systems*, volume 1762 of *Lecture Notes in Computer Science*, pages 31–49. Springer-Verlag, 2000.

[BP00b] Joachim Biskup and Torsten Polle. Decomposition of object-oriented database schemas. Submitted for publication, 2000.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[Cod90] E. F. Codd. *The Relational Model for Database Management, Version 2.* Addison-Wesley Publishing Company, Reading, MA, 1990.

[IW94] Minoru Ito and Grant E. Weddell. Implication problems for functional constraints on databases supporting complex objects. *Journal of Computer and System Sciences*, 49(3):726–768, 1994.

[KLW95] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[Pol00] Torsten Polle. *On Representing Relationships in Object-Oriented Databases.* PhD thesis, Universität Dortmund, FB Informatik/LS6, D-44221 Dortmund, 2000.

[Rei80] R. Reiter. Equality and domain closure in first-order databases. *Journal of the ACM*, 27(2):235–249, 1980.

[Tha93] Bernhard Thalheim. Foundations of entity-relationship modeling. *Annals of Mathematics and Artificial Intelligence*, 1993(7):197–256, 1993.

[Wed89] Grant E. Weddell. A theory of functional dependencies for object-oriented data models. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *Proceedings of the 1st Deductive and Object-Oriented Databases (DOOD '89)*, pages 165–184, Kyoto, Japan, 1989. Elsevier Science Publishers (North-Holland).

[Wed92] Grant E. Weddell. Reasoning about functional dependencies generalized for semantic data models. *ACM Transactions on Database Systems*, 17(1):32–64, March 1992.