Integrating schema integration frameworks, algebraically*

Technical Report CSRG-583
Department of Computer Science,
University of Toronto, 2008

Zinovy Diskin Steve Easterbrook Renée Miller
{zdiskin, sme, miller}@cs.toronto.edu

Abstract

The paper presents a framework, in which the main concepts of schema and data integration can be specified both semantically and syntactically in an abstract data-model independent way. We first define what are schema matching and integration semantically, in terms of sets of instances and mappings between them. We also define a schema matching and integration syntactically, and introduce a procedure (the how) for computing the integration of matched schema according to the syntactical definition in fairly abstract terms. The main theorem of the paper states that the result of syntactical integration satisfies the semantic definition and, hence, does produce what we really need. We show how this framework unifies the approach taken in at least five other schema integration proposals and fills in some important gaps in these proposals. Viewed in a broader perspective, the framework developed in the paper integrates the syntactical and semantical sides of model management and, particularly, reveals a remarkable duality between them. The results of the paper can then be seen as an (important yet) particular case of this general duality theory.

*Supported by Bell Canada through the Bell University Labs, NSERC and Ontario Centres Of Excellence

Contents

1 Introduction

Schema integration (lately, schema/model merge) is the problem of building a *global* data schema from a set of *local* schemas (usually with overlapping semantics) to provide the user with a unified view of the entire dataset. The goal is to give the user of the global schema an illusion of a single dataset specified by a single schema. Schema integration appears in many metadata management tasks, e.g., view integration for database design, building mediated schemas for data integration and warehousing, merging ontologies and several others. It is a well-known research topic with an enormous literature. Early works are surveyed in [BLN86, SL90], and references to more recent work can be found in [Len02, PB03]. Yet a sufficiently general, theoretically sound and practically applicable solution still seems to be missing from the literature.

1.1 The problems

Three inter-related aspects of the problem contribute to its complexity. The first is structural conflict between local schemas. The same data can be viewed quite differently by different users and hence may be specified differently in local schemas. For example, one schema may have an attribute fullName while the other has attributes f-name and l-name (first and last names respectively). Or, an enumeration-valued attribute in one schema might correspond to a subclassing hierarchy in another schema (Fig. 1). Or, what is an entity for one user, may be a relationship for another, and an attribute for the third (Fig. 2). For integration, these conflicts must be precisely specified and then reconciled. Though many particular approaches and languages for this purpose have been proposed, a simple yet general definition of what is a structural conflict is missing from the literature.

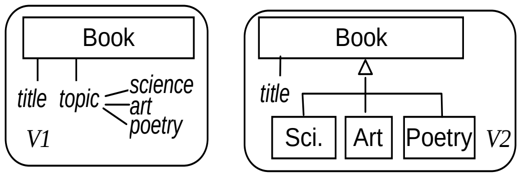


Figure 1: Structural conflict/relativism: two views of the same data

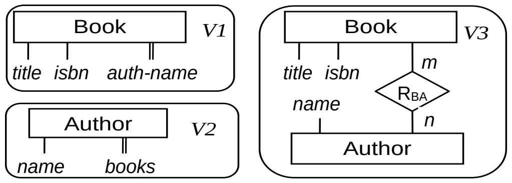


Figure 2: Semantic relativism: three views on the same data

The second source of complexity is extreme heterogeneity of the modern metadata management. Local schemas are usually built in different data models (relational, XML, ER, UML...), and new domain-specific languages appearing on the stage continue to increase the diversity. A typical approach to the problem is first to translate local schemas into some universal data model $\mathcal{U}$, and then integrate them within it. A few questions immediately arise. Is $\mathcal{U}$ sufficiently expressive to accurately represent local schemas without loss of information? If so, how natural are these translations, and do they require significant restructuring? This question becomes especially important if local schemas are actually populated with data. A particular work on schema integration typically argues that the chosen data model is sufficiently universal for "practical purposes", without providing any formal or semi-formal arguments to support such a claim, nor estimating the real scope of universality. The only justification of these choices for $\mathcal{U}$ is common sense, supported by a series of examples. This approach is perhaps sufficient for well-known models like the relational model, XML, and the ER model, for which rich intuition and experience have been accumulated. Yet for newer models like, say, a new UML-profile, or a new domain-specific data model (which the Web never hesitates to provide) require more precise and formal arguments.

Anyway, the arsenal of modeling constructs in $\mathcal{U}$ must be rich enough to simulate structuring capabilities of the local models, and we return again to the issue of structural and semantic conflicts between the views. To capture the diversity of conflicts, languages for specifying view overlapping become complicated and hard to use. Moreover, the algorithms for schema merging are also very complex because they must address each type of conflict and manage it correspondingly.

Finally, most current frameworks for schema integration lack *declarative semantic* definitions of what schema matching and merging are. An integration framework usually provides a syntactical algorithm for schema merging based on given correspondences, and justifies it by some reasonable arguments. However, in the absence of a semantically-based declarative definition, one cannot be sure that one's merge algorithm is sound and computes exactly what one needs. In fact, even semi-formal descriptions of the schema merge problem in terms of database states/schema instances are uncommon and certainly not widely shared among approaches. An excellent early work [BC86] does address semantics but seems to be a notable exception rather than a rule.[1]

[1]Paper [BDK92] provides a *syntactic* declarative definition of merged schema as the least upper bound in the lattice of schemas ordered by their information contents, and [AB01] extends it by working with a graph of schemas and schema mappings rather than a lattice. However, neither of these works takes derived schema elements (queries) into account, which significantly lessens their practical applicability.

1.2 The causes

Overlapping between local schemas is a semantic phenomenon, and the problem of schema integration is inherently semantic. The crucial observation is that the query mechanism plays a major role in reconciling conflicts. This is evident for the schemas in Fig. 1, and in the paper we will show that views in Fig. 2 can also be reconciled with a few simple but *graph-based* query operations. However, the lack of a precise algebraic machinery for specifying queries to semantically rich models has led to ad hoc approaches. For example, the schema integration approach of [PB03] relies on user provided *expressions* for conflict resolution. In general, modern database theory brings a schema integration practitioner to the following alternative. Either one works with the precisely defined and well-understood, but semantically inflexible relational model and relational algebra. Or one uses graphical and semantically rich models like ER-diagrams with various enrichments, whose semantics is intuitive rather than formal, and whose query mechanism is not elaborated. For most practical situations, the second alternative is preferable, and current schema integration methodologies result in complicated solutions lacking precise semantic foundations.

1.3 The approach

The goal of this paper is *not* to present yet another semantic data model and a schema integration algorithm. Our aim is to define a general *specification framework*, where the major ingredients of schema integration – matching, merging and (we will see) normalizing the merge – are clearly specified and supplied with precise formal semantics. In a sense, we aim to build a clear ontology of schema integration procedure, in which different integration steps are clearly separated so that each step is realized with its own methods and tools. In particular, with a properly addressed match, the very merge is a sufficiently trivial automatic algebraic procedure.[2]

[2]The result is always a well-defined schema but it may turn out to be inconsistent, which means that the local schemas are globally inconsistent. For poor semantic models, inconsistency

Technically, the approach is based on the machinery of *graphs with diagram predicates (dp-graphs)* introduced in [DK03]. We define a generic notion of dp-graph, while the signature of diagram predicates is user-defined. (The approach is reminiscent of the way that first-order logic provides the mechanism to build formulas over a given signature of predicate symbols). Some diagram predicates can actually define operations; for example, a ternary predicate add(x,y,z) can be made equivalent to an operation x=y+z. This gives us a framework for query language definition within dp-graphs. The reader may think of a framework where the user can define a sort of relational algebra suitable for applications.

In a nutshell, our integration strategy is a sequence of the following steps.

1. Given a set of local schemas, design a predicate signature rich enough to provide translation of the local schemas into dp-graphs. In general, this is a heuristic procedure but if semantics of the local schemas is well understood, design and translation are not problematic.

2. Match schemas and specify the results by a set of equations between queries against the local schemas. In fact, any correspondence can be specified in this way if the query language is rich enough. To say it briefly, *schema matching is query discovery and equating.*

3. Given a set of equations, run the merge algorithm, which essentially produces the disjoint union of the local dp-graphs augmented with derived elements, and factorizes it by the equations.

   The algorithm also carries the predicates declared in the local dp-graphs to the merged dp-graph. Some of these predicates are query specifications introduced during the matching phase; it means that the merge contains derived elements. Other predicates are constraints declared in local schemas.

4. Run a model checker/theorem prover to check and analyze consistency of the merged constraints.

5. If the merged dp-graph is consistent, normalize it, that is, remove derived elements (as much as possible) to reduce redundancy.

can be automatically detected. For practically interesting semantic models, inconsistency can be immediately detected in simple cases but in general should be checked with the corresponding tools: theorem provers and model checkers. Consistency may be algorithmically undecidable [Con86] or undecidable due to the expressiveness of the constraints considered [**?**]. Thus, the result of schema merge should be checked for consistency.

1.4 Some highlights

- The language of dp-graphs is provably universal: we show in the paper that any data schema with formal semantics can be translated to a dp-graph.[3] Semantics for dp-graphs is truly compositional: it can be presented as a graph morphism from the schema-graph to the universe-graph. In the latter, the nodes are sets (of objects and values), and the arrows are functions between them. Arranging semantics as a morphism allows us to relate operations over schemas (the level of syntax) to operations over sets of instances (semantics).

- The results of matching (query equations) are reified as a new correspondence model/schema equipped with projection mappings into local schemas augmented with necessary derived elements. This gives us a universal pattern for recording all the information necessary for schema merging in a compact way.

- We provide a *declarative* syntactic definition of what is schema merge, and present it as a (graph-based) algebraic operation. We also provide a *semantic* (that is, instance-based) declarative definition of merge. A proof that these two declarative constructions define the same construct (up to renaming of elements) is presented in [Dis06].[4]

- The result of merge contains derived elements and hence can be restructured by making some

[3]We also explain the meaning of provability in this context.

[4]The proof is unnecessarily complicated. A short transparent proof will be presented elsewhere/

of the derived elements basic and original basic elements derived. We call schemas differentiated by this choice *der-equivalent*. We show in the paper that the correspondence schema as well as the merge schema are determined up to der-equivalence.

Some parts of the framework outlined above are novel or may seem novel, but on a whole it is not a radical departure from the existing approaches and ideas. For example, the dp-graphs can be seen as a (far reaching) generalization of the familiar, but now almost forgotten, functional data model [Shi81]. As for integration as such, almost all parts of the picture are known to the community but in a implicit or fuzzy way. Table 1 makes this observation explicit. The rows present the main actors and the columns refer to several well known articles. For each row there is at least one column with a positive mark (though often with reservations). The framework presented in the present paper is thus a sort of closure: we formulate the ideas in precise terms, fill-in the gaps and integrate the pieces into a coherent framework for schema integration.

The rest of the paper is structured as follows. In the next section we briefly survey the existing ideas and works. Section 3 presents the essence of the integration procedure in our approach; the goal is to give the reader a general notion while details are presented in the consecutive sections.

2 Background.

Existing approaches to schema merging can be classified along the following dimensions.

2.1 Manual vs. semi-automatic (heuristics vs. algebra)

In early approaches like [DH84, Mot87]), schema integration is performed by schema restructuring, *ie*, by consecutively applying structural operations taken from a predefined collection to the local schemas. Which operations to take and how to apply them is the responsibility of the global schema designer. The entire process is thus essentially heuristic and human-centric, though the designer may be aided by special tools. Another approach, originated in [SPD92, SP92]splits integration into two principally different phases. In the first one, correspondence between local schemas are specified by a set of assertions in a special language of *correspondence assertions*. Then the global schema is built automatically based on the correspondences specified. In modern *generic model management* parlance [Ber03], both phases are considered as operations over schemas and are called, respectively, *model match* and *model merge*. However, there is an important difference between them. Model matching is a heuristic process of discovering correspondences between schemas and, strictly speaking, is not an algebraic operation. For example, in the case of two local schemas S_1 and S_2, the correspondence specification $\mathbf{match}(S_1, S_2)$ is determined not only by the operands S_1, S_2, but by their semantics and other context dependant factors not specified by the schemas. Thus, an expression $C_{12} = \mathbf{match}(S_1, S_2)$ refers to a semi-heuristic process rather than to an algebraic term. This process can be assisted with intelligent tools using ideas from machine learning and natural language, but a certain heuristic component is principally inevitable. The output of this process largely determines the types of discovery tools used. When the correspondence assertions are simple (uninterpreted) lines, the correspondence specification is commonly called a *schema matching* [RB01, MBR01, LSDR07], when the assertions are queries or general GLAV (global-and-local-as-view) constraints with a formal semantics, the specification is typically called a *schema mapping* [MHH00, PVM$^+$02].

	Mot87	BDK92	SP94	CD96	AB01	PB03	FKMP05
Universe of schemas							
Notion of schema	Version of Functional model (FM)	Richer FM-version	Version of ER-model, ERC+	Graph with diagram predicates and operations	Abstract object with structure implicit	Richer FM version	Relational
The universe is seen as	Set	Lattice	Set	Category	Category	Lattice	N/A
Is duality between syntax and semantics addressed?	No	No	No	No	Yes but abstractly: postulated but not explained	No	Yes
Schema matching:							
How interschema corresp. is specified	Schema restructuring	Name coincidence	Correspondence assertions	Arrow span	Arrow span	Arrow span	TGDs
Does derived info play an important role?	Yes	Minimally and implicitly	No	Yes	No	Implicitly, via expressions on mapping elements	Yes
Declarative syntactic def.	No	No	No	Yes	Yes	Yes	Yes
Declarative semantic def.	No	No	No	Yes	No	No	Yes
Schema merging:							N/A
Declarative syntactic def.	No	Yes, alg.operation of *lub*	No, just an algorithm	Yes	Yes, alg. operation of colimit	Yes, alg.operation of *lub*	
Does derived info play an important role?	Yes	Minimally and implicit. (via closure cond.)	No	Yes	No	Implicitly, via expressions	
Is merge determined up to der-equivalence?	No	No	No	Yes	No	Somewhat (partially, implicitly)	
Declarative semantic def.	No	No	No	No	No	No	

Table 1: Survey table

In contrast to the semi-heuristic match operator, when correspondences between schemas are given by some specification C_{12}, the global schema G can potentially be computed automatically. That is, it is uniquely determined and can be algebraically written as

$$G = \mathbf{merge}(S_1, S_2, C_{12}), \tag{1}$$

where **merge** refers to a ternary operation $\mathcal{S} \times \mathcal{S} \times \mathcal{C} \to \mathcal{S}$ with $\mathcal{S}$ denoting the universe of schemas and $\mathcal{C}$ the universe of correspondence specifications.

An important question of the automatic approach is what are the objects of the universe $\mathcal{C}$? In early work on automatic integration [SPD92] they are statements in a special language of correspondence assertions tailored for a particular data model ERC+, an enriched version of ER-diagrams. More recently, GLAV schema mappings, most commonly specified as formulas of first-order logic adapted for the relational data model are used [Len02, FKMP05]. A disadvantage of these specifications for schema merging is that the third operand C_{12} is conceptually and structurally different from the first two operands and, thus, **merge** needs to handle two very different artifacts in a coherent way.

A different and prominent idea proposed in [CD96, AB01, PB03] is to reify the correspondence specification as a schema, say, S_0, equipped with two functions (projections) $f_{0i}\colon S_0 \to S_i$, $i = 1, 2$ into local schemas. In mathematical category theory, a set of functions/arrows with a common source is called a *span* and thus a correspondence specification is a span

$$C_{12} = \Big(S_1 \xleftarrow{f_{01}} S_0 \xrightarrow{f_{02}} S_2 \Big).$$

Schema S_0 is called the *head* of the spans and functions f_{0i}, $i = 1, 2$ are *arms*. In modern metadata management literature, schemas are often called models and the entire span configuration as above a *model mapping*. Also, the projection functions are often left implicit.

2.2 How to specify and manage structural conflicts

Whatever language/format is used for specifying correspondences, the main question is whether it is expressive enough to capture overlapping of local schema semantics in precise terms. Moreover, if we speak about an automatic phase of schema merge, the language must be formal. The issue is non-trivial because of *structural* (sometimes called *semantic*) conflicts between local views.

Obviously, for automatic schema merge we need a precise taxonomy and specification of possible conflicts. In [SP92], a taxonomy distinguishing *semantic, descriptional, structural* and *heterogeneity* conflicts is presented. However, their description is largely informal and is tailored for a particular data model ERC+. Paper [PB03] elaborates the issue further and presents a more precise taxonomy in terms of a much more universal tree-based schema format. They distinguish *representation*, *meta-model* and *fundamental* conflicts. The former address the possibility of different representations of the same data (structural conflict) while the latter address the integrity of the merged schema after absorption of the information from the local schemas. The meta-model conflicts are conflicts of translations between particular schema languages and the universal language used in [PB03]. Very simple conflicts are well managed in [PB03] by the very span format for correspondence; for more complex conflicts, a mechanism of algebraic *expressions* attached to mapping elements is used. Unfortunately, the details of the expression language are not specified, and it decreases the precision of the machinery making it partially heuristic.

2.3 Semantic justification of syntactic procedures

In the present paper, by *semantics* we mean considering database states or else *schema instances* in a explicit and precise way. That is, if S is a schema, we consider its set of instances $\boldsymbol{inst}(S)$, functions defined on them and operations over them. Note that the term "semantics" is often used for talking about purely syntactical requirements intended to address semantic issues. In this case, semantics is implicit and (at least because of this) is informal.

The consideration of semantics in a precise way is important for schema integration. Consider the following questions. Is a given pool of restructuring operations, or a given correspondence assertion language expressive enough for specifying all possible conflicts between local schemas, or all possibilities of their overlapping? Can a given data model serve as a common universal model for the local models, or to what extent is it universal, or how natural are translations from the local models into this common data model? These questions are essentially semantic and can be answered only in a formal semantics framework. However, formal semantics is rarely addressed in schema integration approaches (with a few notable exceptions [BC86]), particularly those based on informal, but expressive, semantic data models.

The majority of schema integration approaches include clear syntactical procedures that are motivated by reasonable, but informal, semantic considerations. Thus, semantics in the sense above is not addressed and the question of whether the result of a syntactically defined procedure is semantically sound, that is, whether we compute what we really need to compute, remains more a subject of belief rather than formal verification. In many approaches there are no declarative definitions of what is the merged schema at all; hence the merge algorithm is simultaneously a definition of what is computed. A declarative (but still syntactical) definition of what is schema merging appeared first in [BDK92]. They organize the universe of schemas into a set partially ordered by inclusions between schemas, and show that this poset is a lub semi-lattice (that is, the least upper bound of two schemas always exists). Paper [PB03] significantly developed this idea by applying it for a much richer type of schemas and for a much richer set of possible correspondences between them. Unfortunately, their framework becomes less precise as soon as correspondence specifications between schemas involve expressions attached to mapping elements.

On the other hand, research on the data exchange problem is properly semantically based from the very beginning [FKMP05]. They first define the notion of mapping between schemas and its derivatives semantically and then consider necessary syntactical means to specify these notions. There are two obstacles in direct use of these results for schema integration: (i) the context of data exchange is different and (ii) the formalisms is tightly connected to the relational data model.

2.4 Summary of Schema Integration Approaches

In Table 1, we present a summary of our observations about the state-of-the-art in schema integration. We include six papers on schema integration methods, along with a paper on data exchange. Although the last paper does not include a schema merging operation, it is notable for including a declarative semantics for schema matching. It is also relevant to the approach we will be presenting because it explicitly recognizes that structural (semantic) conflicts can be resolved by explicitly representing derived information as queries.[5] This same observation was made much earlier by Motro [Mot87] and [CD96], but omitted from later integration work.

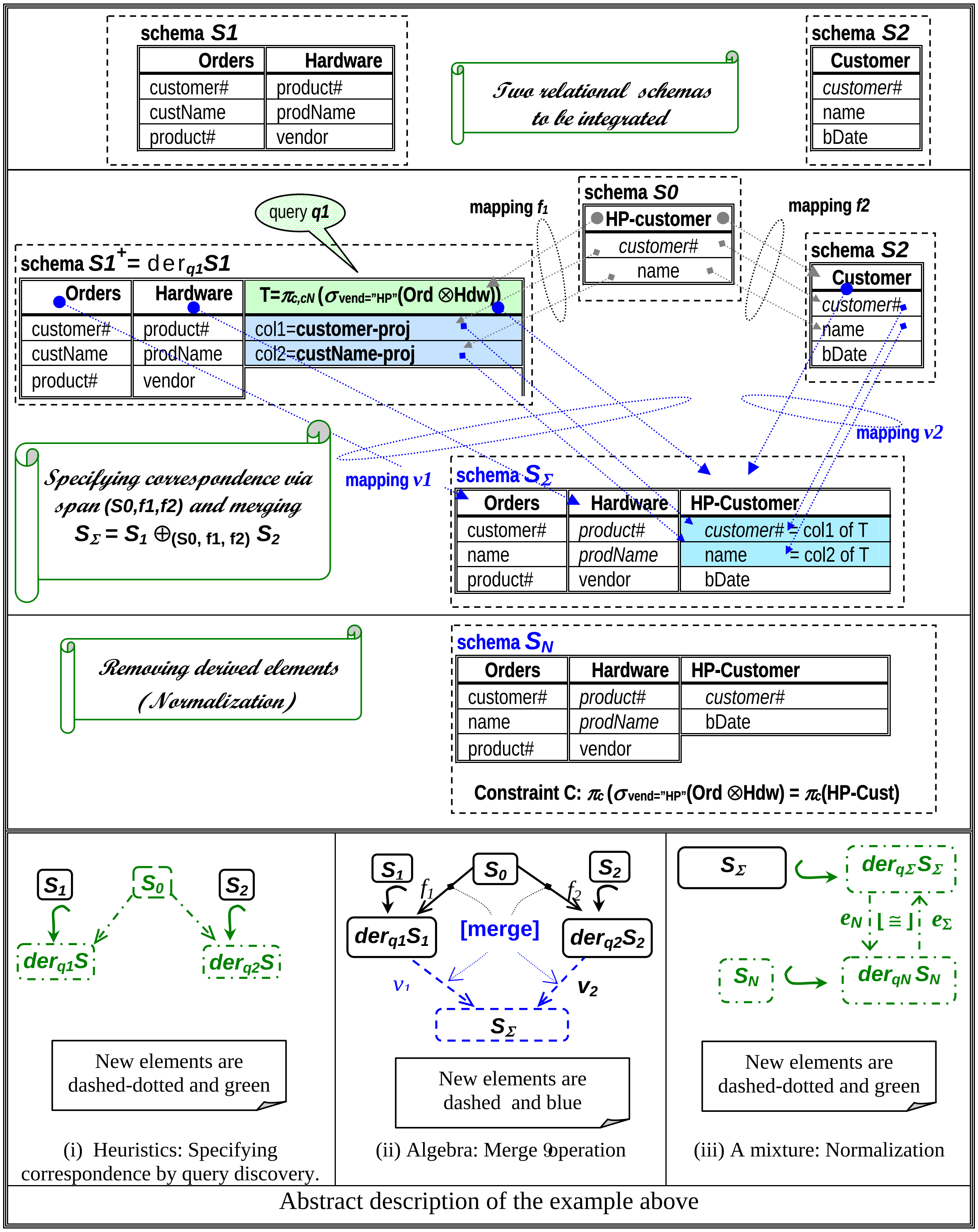
As we have noted, all of these approaches have limitations in some important aspects. Our goal in this paper is to present a schema integration approach without these limitations and in a conceptually clear way. In the next section, we present an example of schema integration of two relational schemas. We use the example to show the main phases of our approach and the main problems to be solved.

3 Sample scenario

3.1 Example of relational schema integration

Consider two simple relational schemas in the top part of Fig. 3. The schema S_1 consists of two tables, Orders and Hardware, each having three columns whose names are written under the table names. The second schema consists of one three-column table

[5]Indeed an early paper on discovering schema mappings for data exchange was called "Schema Mapping as Query Discovery" [MHH00].



schema S1
Orders: customer#, custName, product#
Hardware: product#, prodName, vendor
Two relational schemas to be integrated
schema S2
Customer: customer#, name, bDate
query q1
mapping f1
schema S0
HP-customer: customer#, name
mapping f2
schema S1+ = der_q1 S1
T=π_c,cN (σ_vend="HP"(Ord ⊗Hdw))
col1=customer-proj
col2=custName-proj
Specifying correspondence via span (S0,f1,f2) and merging S_Σ = S_1 ⊕(S0, f1, f2) S_2
mapping v1
mapping v2
schema S_Σ
Orders: customer#, name, product#
Hardware: product#, prodName, vendor
HP-Customer: customer# = col1 of T, name = col2 of T, bDate
Removing derived elements (Normalization)
schema S_N
Orders: customer#, name, product#
Hardware: product#, prodName, vendor
HP-Customer: customer#, bDate
Constraint C: π_c (σ_vend="HP"(Ord ⊗Hdw) = π_c(HP-Cust)
S1, S0, S2, der_q1S, der_q2S
New elements are dashed-dotted and green
(i) Heuristics: Specifying correspondence by query discovery.
S1, S0, S2, f1, f2, der_q1S1, der_q2S2, [merge], v1, v2, S_Σ
New elements are dashed and blue
(ii) Algebra: Merge 9operation
S_Σ, der_qΣ S_Σ, e_N, [≅], e_Σ, S_N, der_qN S_N
New elements are dashed-dotted and green
(iii) A mixture: Normalization
Abstract description of the example above

Figure 3: Example of relational schema integration. [Derived elements are shaded (and blue in color print) . In long expressions, names of tables and columns are abbreviated by a few letters.]

Customer. Suppose we know that the set of customers referred to by schema S_2 is exactly the set of those customers for schema S_1, who ordered products of vendor “HP”. This latter set can be computed by an evident query q_1 specified in schema $\mathsf{der}_{q_1} S_1$ in Fig. 3.

Our first step is to specify the correspondence between schemas in a formal and sufficiently general way. To this end, we augment schema S_1 with a derived table T[col1,col2]: see Fig. 3 where the derived elements are shaded. Table T is computed by query q_1 as specified in the Figure. We denote the augmented schema by $\mathsf{der}_{q_1} S_1$ or ${S_1}^+$. Then we introduce a new schema S_0 with the only table HP-Customer and show how this table is represented in the local schemas. This work is done by schema mappings $f_0 j\colon S_0 \to {S_j}^+ = \mathsf{der}_{q_j} S_j$ $(j = 1, 2)$, which map elements of the middle schema to the corresponding elements, either basic or derived, of the local schemas. In our particular example, ${S_2}^+ = S_2$. Thus, the correspondence information is specified by the following configuration of schemas and mappings between them

$$\mathcal{C}_{12} = \Big(\mathsf{der}_{q_1} S_1 \xleftarrow{f_0 1} S_0 \xrightarrow{f_0 2} \mathsf{der}_{q_2} S_2\Big).$$

We will call such a configuration a *span.*

The span $\mathcal{C}_{12}$ can be thought of as a set of equations between local schema elements like *col1*$@{S_1}^+$ = *customer#*$@S_2$ and *col2*$@{S_1}^+$ = *name*$@S_2$, where expression $e@S$ means that $e \in S$. These equations are necessary for the merge algorithm to eliminate duplication of elements in the merged schema.

The correspondence span we have built has a clear semantic meaning. Note that mapping f_{01} defines a view to schema S_1 (with schema S_0 being the view schema). Semantically, it means that for any instance $I \in \boldsymbol{inst}(S_1)$, the result of view/query execution is defined as $[\![f_{01}]\!](I) \in \boldsymbol{inst}(S_0)$. Here we use the following notation: if X is a syntactic notion, a schema or mapping between schemas, then $[\![X]\!]$ is its semantic meaning, the set of instances or the mapping between these sets respectively. The same is for mapping f_{02}, in our case trivial. Thus, the semantic meaning of the span declaration is the following equality: $[\![f_{01}]\!](I_1) = [\![f_{02}]\!](I_2)$, where I_j is an instance of schema S_j for $j = 1, 2$. That is, S_0 specifies the shared information in S_1 and S_2.

The natural next step is to merge the augmented local schemas disjointedly, and then glue together the elements that are declared equal in the correspondence span.The resulting schema S_Σ will be a disjoint union of the three components:

$$(2) \qquad S_\Sigma \cong ({S_1}^+ \setminus S_0) \uplus S_0 \uplus ({S_2}^+ \setminus S_0).^6$$

In addition, we obtain mappings (views) between the local schemas and the merge, v_1 and v_2 in the figure. Thus, the result of integration is a configuration of schemas and mappings $\mathcal{S} = (S_\Sigma, v_1, v_2)$ specified in Fig. 3. We will call such configurations *sinks.*

If schemas in the correspondence span contain derived elements, the merge schema will also contain derived elements. To finish integration, we may like to remove from S_Σ derived (shaded) items. In general this process is not trivial since removal derived items may violate some structural requirements to schemas. For instance, in our example, removing the derived column *cutomer#* will leave the table HP-Customer without its primary key. Hence, we forced to keep it in the schema but then add to it a corresponding integrity constraint: this is Constraint C in Fig. 3, which asserts equality of the results of two queries.

Thus, an important final part of the integration procedure should be a special procedure of schema *normalization.* The latter is aimed at finding some schema S_N with minimal redundancy but der-equivalent to the merge schema. That is, each element of the merge schema S_Σ can be derived from S_N (no information is lost) and conversely, each element of S_N is derivable from S_Σ (nothing extra is acquired). It is easy to see that our schema S_Σ can be further normalized but a detailed discussion goes beyond the goals of this paper but has been addressed by others [AH88].

[6] This would be a quite precise description if schemas were sets of elements. However, relational, XML and other schemas used in practice are much richer structures, and the merge procedure must be compatible with it. Later we will address the issue.

3.2 Abstract arrangement

The diagrams in the bottom part of Fig. 3 present an abstract view of the procedures we used in the example. This view is based on abstract graphs and operations over them. Nodes of these graphs denote schemas (considered as structured sets of elements) and arrows are mappings between them (preserving the structure); hooked arrows denote inclusions. In diagram (ii), label join denotes the graph-based analog of the familiar lattice-theoretic operation of taking the least upper bound (cf. [BDK92]). This analog is defined as follows. Mappings $v_i\colon S_i \to S_\Sigma$, $i = 1, 2$ say that all the information contained in the local schemas is contained in the merged schema as well without duplication. The latter condition is provided by the commutativity of the diamond formed by the arrows: for any element $e \in S_0$, equality $e.f_{01}.v_1 = e.f_{02}.v_2$ holds in S_Σ. Thus, information of local schemas is properly passed to S_Σ and no information is lost (S_Σ is an upper bound). To formulate that nothing extra is acquired (S_Σ is the least upper bound), we take an arbitrary schema S' together with mappings $v'_i\colon S_i \to S'$, $i = 1, 2$ such that the diamond $(f_{01}, v'_1, f_{02}, v_2)$ is commutative. That is, schema S' also properly (without duplication) contains the information of the local schemas (another upper bound). The minimality property of S_Σ then means that for any such S' there is a uniquely defined mapping $!\colon S_\Sigma \to S'$ such that every diagram formed by all the arrows involved is commutative. In mathematical category theory such an operation on objects is called *colimit*. Thus, it is *reasonable to define* the result of merging schemas matches by a span as the colimit of the span. We will use the more familiar term *(arrow) merge* as a synonym for colimit. Note that the result of [**merge**] is a *configuration* of schemas and mappings, a *cospan* or *sink* $\mathcal{S} = (S_\Sigma, v_1, v_2)$, rather than the merged schema itself.

A well-known result of mathematical category theory says that for a wide class of meta-models, the classes of schemas defined by those metamodels are closed under arrow merge. Moreover, for this class of metamodels, merge defined above declaratively, can be defined constructively as well. Roughly, a merge is computed by taking disjointed union of the two schemas followed by factorization according to the "equations" specified by the correspondence span $\mathcal{C}_{12} = (S_0, f_{01}, f_{02})$. A typical algorithm of this sort for the case of graphs is presented in [SE05]. It can be immediately generalized for any schema consisting of sets of elements and mappings between them [BW95]. In this way we obtain an effective procedure for computing merged schemas defined w.r.t. their information content. Finally, it can be proved that the declarative syntactical definition above is equivalent to a reasonable semantic definition of merge in terms of instances [Dis06].

3.3 Discussion

We have seen that schema integration consists of three consecutive steps: matching, merging, and normalizing the merge. The first and the third are context-dependant heuristic procedures (that may be assisted by tools but not fully automated), the second step is an algebraic operation and can be entirely automatic. The key to a proper integration is not in the merge itself but in a proper specification of their overlapping/correspondences often called *structural conflicts*. Below we call them conflicts.

A major integration issue is thus how to classify and specify conflicts between schemas: a few attempts were made in the literature, most notable and systematic are [SPD92, PB03]). These classifications are quite complicated and lead to considerable complexity of the merge algorithms. A major contribution of the present paper is to show that all conflicts between views can be uniformly described as in our example above: a schema element basic in one view can be derived in another view (and vice versa). In general, a conflict appears as the "sameness" of two elements, $e_1 \simeq e_2$, where e_i denotes the result of applying a query q_i to schema S_i, $i = 1, 2$. Then the merge algorithm itself is a purely algebraic procedure and is simple.

The first step towards drawing this base is to avoid syntactical idiosyncracies of relational or other particular data models and consider the integration problem on the basis of what we call a *universal* data model; the query language is also included as part of the model. A universal model must (i) be tailored for specifying semantics in as direct a way as possible, (ii) be semantically rich to allow specification of a wide class of conflicts, (iii) allow for natural translation from practically used data models like relational, XML or ER and yet, (iv) be syntactically manageable and semantically suggestive. We will present a framework, in which it is possible to build such model, dp-graphs, in the next section. After that, we will show how various conflicts identified in the literature can be specified by query equations over dp-graphs.

4 Universal data model

Dp-graphs were introduced (under the name of sketch) and applied to data modeling in [DK03]. To explain the idea, we begin with an example. It is simple but sufficiently generic to present the language of dp-graphs and its expressive capabilities.

4.1 Example of data definition with dp-graphs

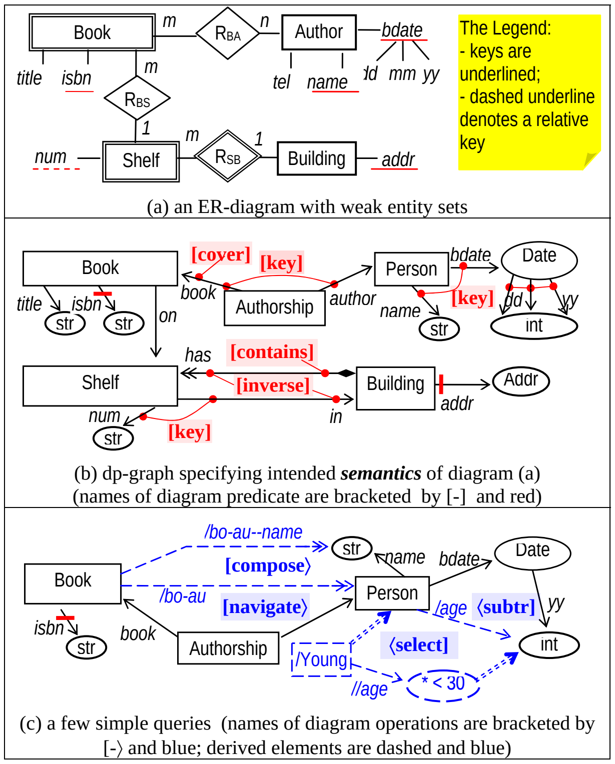
Fig. 4(a) presents a simple ER-diagram. Semantics of its elements is clear from their names but we want to make it precise. Let t be a time moment, or the database/real-world state at moment t. The rectangle nodes give us four sets $[\![\mathsf{Book}]\!]^t$, $[\![\mathsf{Author}]\!]^t$, $[\![\mathsf{Shelf}]\!]^t$ and $[\![\mathsf{Building}]\!]^t$. Attributes are functions $\mathsf{title}\colon [\![\mathsf{Book}]\!]^t \to [\![\mathbf{str}]\!]$, $[\![\mathsf{tel}]\!]^t\colon [\![\mathsf{Author}]\!]^t \to [\![\mathbf{int}]\!]$ *etc*, where $[\![\mathbf{str}]\!]$, $[\![\mathbf{int}]\!]$ are extensions of primitive types, which do not depend on time. Note that domains of the attribute are implicit in ER-diagrams and we used semantics of names to reveal them. Diamond nodes denote relations: $[\![R_{BA}]\!]^t \subset [\![\mathsf{Book}]\!]^t \times [\![\mathsf{Author}]\!]^t$ *etc.* Relations $[\![R_{BS}]\!]^t$ and $[\![R_{SB}]\!]^t$ are of type many-to-one and hence are functions at any moment. Furthermore we will omit the index t keeping it in mind. We will also omit the square brackets if it does not lead to confusion.

Formal explication of what are keys, relative keys and weak entity sets need slightly more accurate formulations, and we proceed to Fig. 4(b). In this diagram, nodes denote sets and arrows are functions, either single-valued (an ordinary head) or multi-valued (double-head). Rectangles denote sets with extension changing in time (classes), and ovals denote primitive types with permanent predefined extension. A family of functions with a common source $f_i\colon X \to Y_i$, $i = 1..n$ is called a *key* if for any two *different* elements of the source, $a, b \in X$, there is at least one function f_i in the family such that $f_i(a) \neq f_i(b)$. Then in the diagram, which is a syntactical object, label [**key**] "hung" on the pair of arrows name and bdate means that at any time moment t, the pair of functions $[\![\mathsf{name}]\!]^t$ and $[\![\mathsf{bdate}]\!]^t$ is a key. Similarly, declaring the pair of arrows author and book to be a key to the node $\mathsf{Authorship}$ actually means that extension of this node is a set of pairs (a, b) with $a \in [\![\mathsf{Author}]\!]^t$ and $b \in [\![\mathsf{Book}]\!]^t$. If a single arrow is a key, we mark it with a short bar crossing the tail.

Note that node Book in the ER-diagram is declared to be a weak entity (double-frame). Suppose that it means that each book participate in at least one $\mathsf{Authorship}$ and hence depends on it (or the corresponding author). Formally, it means that the function $[\![\mathsf{book}]\!]^t$ is surjective (or covering) at any time moment; hence, the label [**cover**] on arrow book.

Since relationships R_{BS} and R_{SB} are of type many-to-one, they are in fact functions and we replace them by arrows $\mathsf{on}\colon \mathsf{Book} \to \mathsf{Shelf}$ and $\mathsf{in}\colon \mathsf{Shelf} \to \mathsf{Building}$. Note the label [**key**] hung on the pair $(\mathsf{num}, \mathsf{in})$ – this the precise formal meaning of declaring the node Shelf to be a weak entity in the sense that its key num is relative to the Building the shelf is in. Note that this semantics of weak entity is quite different from semantics of node Book, and the difference is made explicit in our dp-graph. In the literature on ER-diagrams, there is one more meaning of being a weak entity, which is usually called existence dependency. Namely, in our context, if a Building is deleted from the universe (the database), all its Shelves are also deleted. This is a *dynamic* property of function $[\![\mathsf{has}]\!]^t$, which we make explicit by declaring predicate [**contains**] for arrow has.[7] Formal definition

[7]In addition, as a syntactic sugar tip, we also make the tail of the arrows a black diamond: this notation is borrowed from UML and is now common. Note that usually semantics of black diamond includes not only existence dependency (our predicate [**contains**]) but also non-sharing: every shelf belongs to one and only one building. In our dp-graph this is captured



(a) an ER-diagram with weak entity sets

(b) dp-graph specifying intended ***semantics*** of diagram (a) (names of diagram predicate are bracketed by [-] and red)

(c) a few simple queries (names of diagram operations are bracketed by [-⟩ and blue; derived elements are dashed and blue)

Figure 4: Dp-graphs vs. ER-diagrams

of the property [**contains**] is more intricate than above because it involves specifying *behavior* of function $[\![\mathsf{has}]\!]^t$ when t changes; details can be found in [DK03].

Finally, to express the required semantics of the configuration Shelf - R_{SB} - Building, we have introduced two functions, in and has, representing the same relationship R_{SB}. Hence, these functions are mutually inverse and we must declare this fact in out dp-graph: note label [**inverse**] hung on the respective pair of arrows. The resulting dp-graph in Fig. 4(b) has the following two main features: it does possess a precise formal semantics (which is close to the intended semantics of the ER-diagram), yet it is syntactically transparent and similar to the ER-diagram.

by declaring the arrow in (inverse to has) to be single-valued.

4.2 Querying dp-graphs

A special subclass of diagram predicates are diagram operations. For example, we may consider an operation **navigate**: its input is a pair of functions with a common source (multi-relation), and the output is the function of navigating the relation from its one participant to the other. Syntactically, declaring such operation looks like shown in Fig. 4(c). Semantically it means that having a relation

$([\![\mathsf{Authorship}]\!]^t, [\![\mathsf{book}]\!]^t, [\![\mathsf{author}]\!]^t)$

with two projection functions to the classes Book and Author, we navigate the relation and produce a new (in general, multi-valued) function

$/\mathsf{bo\text{-}au}\colon [\![\mathsf{Book}]\!]^t \to [\![\mathsf{Author}]\!]^t$.

More accurately, /bo-au/ is the name of the new function denoted in the dp-graph by a new arrow /bo-au, and the very new function is its extension $[\![/\mathsf{bo\text{-}au}]\!]^t$. We denote derived elements by dashed blue lines and their names are prefixed with slash (the latter notational tip is borrowed from UML).

Having a pair of functions with the target of the first being the source of the second, e.g., $[\![/\mathsf{bo\text{-}au}]\!]^t$ and $[\![\mathsf{name}]\!]^t$ in Fig. 4, we can compose them and produce a new function. Syntactically we denote this new function by an arrow /bo-au-name. Semantically, its extension $[\![/\mathsf{bo\text{-}au\text{-}name}]\!]^t$ is the result of executing the operation **compose**. The right lower part of the diagram shows a query specification, whose execution would produce a set of Persons younger than 30.

Thus, queries against dp-graphs are diagram operations: they input and output configurations of sets and mappings (specific and predefined for a specific operations). Technical details of how syntax and semantics of such operations is formally defined can be found in [Dis96]. We emphasize that we do not provide an a priori fixed query language. It is the user's responsibility to define her own query language in accordance with dp-graphs' syntax, and supply it with execution procedures.

4.3 Universality statement.

Thesis. Let S be some data schema (relational, XML, ERD, yours favorite one). If semantics of S

can be somehow formalized, then there is a dp-graph (graph with diagram predicates) G such that sets ***inst***(()S) and ***inst***(()G) are naturally isomorphic.

"Proof"/Justification. Let X be some notion, eg, semantics of S. Formalizability means that X can be expressed in some formal set theory developed in modern mathemtics. There are a few such theories, which are roughly equivalent between themselves. One of the most handy and convenient in applications is the higher-order type theory, HOTT, presented, e.g., in [LS86]. Thus, we may define formalizability of X as the possibility to present X as a HOTT-theory.

In part II of [LS86], the following results are carefully proven. Any HOTT-theory generates a special algebraic structure called *topos*. Conversely, any topos can be presented as a HOTT-theory. Moreover, the notions of HOTT-theory and topos are equivalent in some technical sense. Thus, we may say that formalizability of X means the possibility to present X algebraically as a topos.Finally, any topos is nothing but a dp-graph in a specific signature of diagram predicates (in fact, operations), see e.g. [FS90] for a precise but "pictorial" definition. We conclude that formalizability of X means that X can be encoded by a dp-graph.[8]

4.4 FAQ about dp-graphs

How natural are translations of local schemas in different data models to dp-graphs? If semantics of a local schema is well understood in precise (better, formal) terms, translation to a dp-graph is easy because dp-graphs are nothing but specifications of schema semantics.

How complex is the vocabulary of basic concepts for dp-graphs? Any dp-graph consists of elements of only three types: nodes, arrows and diagram predicates. In addition, some of the elements can be declared derived, which means that the corresponding diagram predicate is actually an operation.

[8]Speaking more accurately, we have mentioned three formal models, *ie*, three formal definitions, of the notion of formalizability. What is mathematically proven is that they are all equivalent.

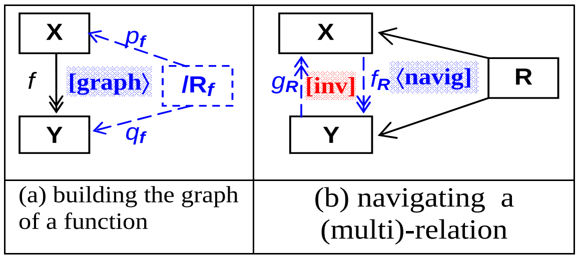

X
p_f
f
[graph⟩
/R_f
Y
q_f
(a) building the graph of a function
X
g_R
[inv]
f_R
⟨navig]
R
Y
(b) navigating a (multi)-relation

Figure 5: The representation conflict of dp-graphs

How complex are possible structural conflict between different dp-graphs representing the same data? We will show in the next section how different representation conflicts can be managed by queries. Since the vocabulary of structural element in dp-graphs consists of only nodes and arrows, there is only one type of structural conflicts. To wit: data seen as a set (node) by one dp-graph, is seen as a function (arrow) in another dp-graph. This conflict is resolved by two queries/operations as shown in Fig. 5.

5 Schema matching as query discovery and equating

In this section we show that the diversity of conflicts between schemas can be reduced to basically the only one: a query against one schema and a query to another schema produce the same result (up to canonic isomorphism). We will call such statement a *query equation*. The idea appeared in [CD96] and then in [MBHR05] but it seems its full potential is still not realized. The most elaborated taxonomies of conflicts reported in the literature are those in [SP92] and [PB03]. Since the latter claims that its taxonomy and merge algorithm subsumes those in [SP92], we will carefully consider how conflicts considered in [PB03] can be described by query equations. To ease references, we will call the taxonomy and merge procedure specified in [PB03] Vanilla – the name of the tool implementing them. Also, in this section we use terms schema and graph interchangeably.

5.1 Vanilla's constructs via dp-graphs

5.1.1 Preliminaries.

In this section we apply general patterns for specifying inter-schema correspondences (section 3.2) in the dp-graph framework. For dp-graphs, the important condition of preserving the structure by inter-schema mappings (projections of correspondence spans and injections of local schemas into the merged ones) is formulated as follows.

First of all, mappings must be compatible with incidence relations between nodes and arrows. Given two graphs G, H and a mapping $f\colon G \to H$ between them, we require that if $f(a) = b$ for arrows $a \in G$ and $b \in H$, then

$$f(\Box a) = \Box b \text{ and } f(a\Box) = b\Box, \tag{3}$$

where $\Box *$ and $*\Box$ denote the source and the target nodes of an arrow $*$. We will often call mappings between graphs satisfying this requirement *graph morphisms*. To specify a graph morphism, it is sufficient to specify it for arrows and isolated nodes. For example, the only dotted curly arrow in the right half of Table 2(a') specifies morphism f_{02} as precisely as three dotted arrows specify f_{01} on the left. The second condition requires preservation of diagram predicates. Given two dp-graphs G, H, a graph morphism $f\colon G \to H$ is called a *dp-graph morphism*, if for any group E of nodes and arrows in G labeled by predicate P, the image of this group in graph H, $f(E)$ is also labeled by P. Below we will call morphism mappings as it is a suggestive and natural name for them. We remind that the configuration

$$C_{12} = \Big(S_1 \xleftarrow{f_{01}} S_0 \xrightarrow{f_{02}} S_2\Big).$$

is called a *correspondence span* with schema S_0 the *head* and projection mappings f_i the *legs*.[9]

The left column of Table 2 presents simple examples (taken from [PB03]) illustrating Vanilla's taxonomy of conflicts. In the right column we specify these examples with dp-graphs and mappings between them. We will consider the table row by row.

[9]Warning: in the literature, particularly in Vanilla, spans are often called mappings while the projections are left nameless.

5.1.2 Synonymy

Conflict (a). It is a simple synonymy situation. Both specifications (a) and (a') are practically equivalent yet two points of difference are worth mentioning. (i) For us, attributes are arrows targeted at primitive types and hence attribute domains become explicit. Since schema integration is all about semantics, explicit specifications are preferable. (ii) An equation between elements is set by projection mappings and labeling the corresponding element in schema S_0 with equality symbol is not necessary. The head of the correspondence span is an *ordinary* schema like S_1 or S_2, in which there are neither equality nor similarity labels.Particularly, it allows us to write names of correspondence nodes right on them, which is convenient and helpful in practical work.

Conflict (b). Here the situation is a bit more complicated. Attributes f-name, l-nameof the right schema (S_2) are absent in the left schema (S_1) but can be derived in it. This derivation can be specified either in the left schema itself, or in the correspondence schema. We choose the latter way as it makes comparison with Vanilla easier. Composition of two evident queries q_{11} and q_{12} against the correspondence schema produces two new arrows /f-name and /l-name/ as needed. Now correspondence between the local schemas can be specified by setting projection morphisms as shown.

Remark. All examples in Table 2 are very special in the sense that the local schemas are subsumed by the correspondence schema and hence the latter is simultaneously the merged schema as well. In more detail, what the merge algorithm described in the next section does is "absorbing" (without any damage to the structure!) those parts of local schemas, which are in the range of projection mappings, into the correspondence schema . The name conflicts are uniformly resolved: the correspondence schema names dominate. Since for the examples in the table, projections are surjections (cover their target), local schemas will be entirely absorbed. This consideration helps to interpret the examples.

The result of merge in cell (b') is thus a schema where the local schemas are embedded. In addition to images of the local schemas, the correspondence schema include also new information: specification of query $q_1 = q_{11}; q_{12}$, which relates data instances of the local schemas.

Conflict (c). Semantically, the situation is entirely similar to the above. Instances of the left schema can be derived from those of the right one by executing a query q_2. We again have partially defined projection mappings, which will absorb the local schemas into the correspondence one. The only difference between (b') and (c') is that the former uses query q_1 to derive the right from the left, while (c') uses query q_2 to derive the left from the right. Moreover, it is easy to see that queries q_1 and q_2 are mutually inverse, hence the correspondence schemas in (b') and (c') are *der-equivalent*: they differ only in the choice of which elements are considered basic and which are derived.

A striking difference in Vanilla's view of the two situations is caused by replacing querying in (b') by a new structural relationship, that is, as a new rather than derived information.[10] Not only replacing queries by sub-structural relationships unjustly complicates the taxonomy and hence the merge algorithm, it may create problems later if the original data model does not have a similar structural relationship. For example, SQL does not have a construct of sub-column. Vanilla calls such problems *meta-model conflicts*. Thus, Vanilla's way of processing a typical representation conflict may create another type of conflict.

5.1.3 Homonymy

Non-mapping element (d). Semantics of the situation is this. Schema matching investigation reveals that left and right Persons refer to the same class but the attributes bio, although having the same name in both schemas, actually refer to different concepts: official bio (o-bio) for the left and unofficial one (n-bio) for the right view. Furthermore, it is revealed that a person actually has two bios but the left view is not interested (does not know about?) n-bio while the right view does not concern about o-bio. (This is semantics of (d) as it is described in [PB03]). The correspondence span in cell (d1') directly specifies exactly this semantics: Persons are glued together while bios are not. The fact of having exactly two bios can be captured by requiring the functions $[\![\,\mathsf{o\text{-}bio}\,]\!]$ and $[\![\,\mathsf{n\text{-}bio}\,]\!]$ to be totally defined. Note that each element of the correspondence schema is either in the domain of projection f_{01} or f_{02}, that is, there are ***no*** non-mapping elements in the span. The following consideration explains what is meant by Vanilla's classification of case (d) as non-mapping element. Let us perform merge/join of the span in (d1'). The result is shown in (d2') with black solid lines. Now we can apply the operation of pairing to the two functions, and obtain a new derived function allBio. For any person P, $\mathsf{allBio}(P) \overset{\text{def}}{=} (\mathsf{o\text{-}bio}(P), \mathsf{n\text{-}bio}(P)) \in \mathit{txt} \times \mathit{txt}$, that is, a pair of texts with first component giving the official, and the second the non-official bio. It appears that Vanilla implicitly introduced into correspondence specification a part of the future merge! To finish the case, diagram (d3') shows a der-equivalent restructuring of the merge, where the attribute allBio is basic while the component bios are derived. (Der-equivalence is implied by the fact that operations of pairing and projecting are mutually inverse).

Conflict (e). We again have a case of homonymy. Zip elements of the views are declared equal but their types are different. Hence, after the merge, element Zip will have two types, which violates the integrity constraint of the model itself: types must be unique. Vanilla calls such conflicts fundamental and pays special attention to them. The way of resolving this type of fundamental conflicts in Vanilla is borrowed from [BDK92] and is shown in the lower half of the diagram (e). The intersection of the types is formed and is considered to be the new unique type. There are a few reasons to consider this solution unsatisfactory. First, it is not clear what is to do with Zip-values outside the intersection. If there are no such values, then the views actually use the same type int-and-str, which can be matched from the very beginning. If such values exist, then the conflict is not resolved.

[10]As for similarity elements with expressions attached, cell (c), this is Vanilla's way to manage query specifications. Btw, machinery of expressions is syntactically hard to manage. It is unclear, for example, how to compose mappings carrying expressions or reverse such mappings – this problem was explicitly stated in [Ber03]. It can be resolved with dp-graphs as shown in [Dis05].

Finally, there are possible other operations on types coercing/unifying them and the choice of intersection only is not justified. For example, diagram (e1') presents another view of the conflict in the dp-graph framework.

First of all we note that when attributes are seen as arrows, it is illegal to equate two zips with different target nodes: we remind that equality of two arrows automatically means equality of their sources and targets.[11] Thus zip-arrows cannot be equated. Yet we can equate their sources thus coming to situation in (d1'). It would mean that an address has two zips, one of type **int** and the other of type **str**. That would be an interesting address system but, anyway, if semantics of the case is such, we specify it and merge as shown in cells (d1')..(d3'). However, for address systems, more likely is the situation when different zip types mean that we deal with different address systems, e.g., in US and Canada. The correspondence span specifying this semantics is shown in cell (e1'). The only fact this correspondence spec reveals is that Addr and zip are homonyms and fixes the problem by renaming the Addr nodes (assuming that the arrow names are qualified by the source names, us-zip, ca-zip). Merge is fairly simple as shown in cell (e2'), ignore the blue-dashed part of the graph for a moment. Since two address systems have much in common, it is reasonable to generalize the classes as shown in (e2') (the upper operation [**gen**], double-body arrows denote subclassing). In the simple set-theoretical semantics we consider, the operation is nothing but the disjoint set union).[12]. Similarly, we can generalize the domains into a new domain **int** ∪ **str** (in programming terms, form a variant type). Finally, we define a new generalized attribute /zipin the evident way: it is the **int**-valued zip (left view) for USAddr objects in /Addr and the **str**-valued zip (right view) for CaAddr-objects. Da-graph (e3') presents a der-equivalent restructuring of the merge, in which generalized Address and zip are basic while their US and Ca versions are derived. (Der-equivalence is followed from the fact that operations [**gen**] and [**select**] are mutually inverse).

We see that the situation described in cell (e) before the merge is significantly underspecified. It may be specified either by the correspondence span like in (d1') (an address has two zips) or by the span (e1') (an address has one zip that is either **int** or **str**) or as in (e) (assumed by default in Vanilla). Note that for either-or semantics takes place, it is not necessary that the sources of the attributes were different. We can imagine a single address system but so freely designed that some addresses have **int**-zips and others **str**-zips.Correspondingly, we have different merged schemas. Neither of merges is right or wrong: there are different semantics and respectively there are different merges adequate to them (cf. [MIR93]). Importantly, merges are different because the correspondence specifications for these semantics are different, not because the operation of merge is non-deterministic.

5.2 More on conflicts

Queries vs. constraints. Fig. 6(a) shows a simple conflict resolved by a simple query. However, this would work only if the following two conditions hold. Firstly, we need to know that the class Person in the left view is exactly the class right Person whose age is less than 30. Secondly, the query language must be expressive enough to specify this knowledge in an adequate way. If one of this components is absent, but still we know that the left Person is a subclass of the right Person, we introduce into the correspondence schema the subclassing constraint. This constraint presents a new piece of information captured in neither of local schemas and neither of the projection mappings is defined on it. In Vanilla's terms, this constraint is a non-mapping element. There are also situations when non-mapping elements are data elements (nodes or arrows) rather than constraints.

Books and Authors continued. It is easy to see that with query operations of **navig** or/and **graph** Fig. 5 and function composition $=$, conflicts between views depicted in Fig. 2 can be specified by equations and thus resolved.

The examples we considered show the flexibility

[11]In the model management tool we are implementing [SCE+07], such declaration would be a compile time rather than merge time error.

[12]If the sets were not disjoint, we would specify this in the correspondence span

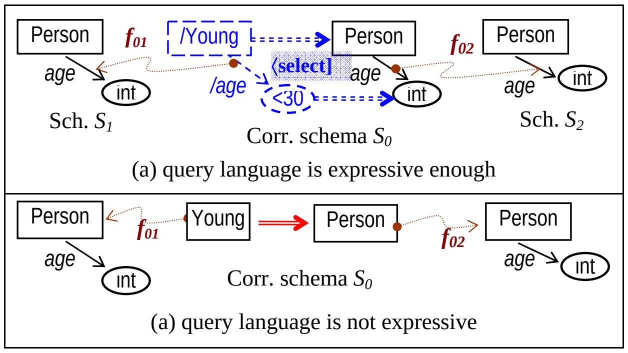


Figure 6: Queries vs. constraints

and convenience of the *Query Discovery and Equating* (QDE) approach in the framework of dp-graphs. Moreover, based on the Universality statement (section 2.3), we can say that any overlapping between views, which can be formally specified, can be specified by QDE in a suitable signature of dp-graphs.

6 Conclusion

The goal of the paper was to analyze the problem of schema integration in precise conceptual and technical terms. To achieve it, we have applied a mathematically sound and powerful semantic model based on graphs with diagram predicates (constraints) and diagram operations (queries), dp-graphs in short. We have carefully analyzed major research works on schema integration, reformulated them in precise terms of dp-graphs, removed inconsistencies and redundancies within and between the approaches, and integrate them into, we believe, a coherent framework. Its main highlights are as follows.

Schema integration consists of three major phases: schema matching, merging and normalizing. Matching is the key to the entire problem because of the infamous problem of specifying overlaping/conflicts between views in a manageable way. We have shown that all types of conflicts identified in the literature can be reduced to the following one universal type: elements basic in one view are derived in another view and can be computed if the query language is rich enough. Thus, specifying inter-schema correspondences is nothing but query discovery and equating. This formulation gives us a universal yet compact specification pattern, which schema matching must fill in, with a clear semantic meaning. In its turn, a syntactically and semantically transparent pattern for inter-schema correspondences gives rise to a syntactically and semantically clear algorithm for schema merging. Particularly, we give a declarative syntactic definition of schema merging as a graph-based analog of the lattice-theoretic notion of the least upper bound. In a companion paper, we give a declarative semantic definition and show that these two definitions are equivalent. Also, having only one type of conflicts between schemas allows us to use a purely algebraic and in fact very simple algorithm for schema merging. However, since the correspondence specification involves derived elements and queries against local schemas, the merged schema also contains derived elements and hence redundancies. It implies a post-merge phase (normalization) aimed at removing redundancies as much as possible. Normalization can be non-trivial; particularly, we have presented an example showing that complete removal of derived elements from the merged schema is not always possible.

To summarize, among the three main steps of integration (match, merge, normalize), the first and the third are heuristic and may be really complex (especially match) while the very merge is pure algebra and simple. We suppose that excessive complexity of merge algorithms proposed in the literature is caused by partial absorbtion of underspecified overlapping and implicit normalization into the very merge procedure. In contrast, in our approach the three phases are explicitly separated and provided with clear specification patterns ensuring their smooth sequential composition. Importantly, clear separation of the entire problem into three well-shaped subproblems allows one to address each of them in a appropriate way and use the appropriate tools.

References

[AB01] S. Alagic and P. Bernstein. A model theory for generic schema management. In *Proc. DBPL'2001*, 2001.

[AH88] S. Abiteboul and R. Hull. Restructuring Hierarchical Database Objects. *Theoretical Computer Science*, 62:3–38, 1988.

[BC86] J. Biscup and B. Convent. A formal view integration method. In *ACM SIGMOD Conf.on Managment of Data*, pages 398–407, 1986.

[BDK92] P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *Advances in Database Technology - EDBT'92*, Springer LNCS # 580, 1992.

[Ber03] P. Bernstein. Applying model management to classical metadata problems. In *Proc. CIDR'2003*, pages 209–220, 2003.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A comparitive analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[BW95] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1995.

[CD96] B. Cadish and Z. Diskin. Heterogenious view integration via sketches and equations. In *Foundations of Intelligent Systems, 9th Int.Symposium, ISMIS '96*, Springer LNAI #1079, pages 603–612, 1996.

[Con86] B. Convent. Unsolvable problems related to the view integration. In *Int.Conf.on Database Theory, Roma*, pages 141–156, 1986.

[DH84] U. Dayal and H. Hwang. View definition and generalization for database integration of a multibase system. *IEEE Trans. Software Eng.*, 10(6):628–644, 1984.

[Dis96] Z. Diskin. Databases as diagram algebras: Specifying queries and views via the graph-based logic of skethes. Technical Report 9602, Frame Inform Systems, Riga, Latvia, 1996. www.cs.toronto.edu/ zdiskin/Pubs/TR-9602.pdf.

[Dis05] Z. Diskin. Mathemtics of generic specifications for model management. In *Encyclopedia of Database Technologies and Applications*, pages 351–366. Idea Group, 2005.

[Dis06] Z. Diskin. Metamodel-independent schema & data integration: Towards joining syntax and semantics in generic model management. Technical Report 2006-522, School of Computing, Queen's University, Kingston, Canada, 2006. `http://www.cs.queensu.ca/TechReports/`.

[DK03] Z. Diskin and B. Kadish. Variable set semantics for keyed generalized sketches: Formal semantics for object identity and abstract syntax for conceptual modeling. *Data & Knowledge Engineering*, 47:1–59, 2003.

[FKMP05] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1), 2005.

[FS90] P. Freyd and A. Scedrov. *Categories, Allegories*. Elsevier Sciece Publishers, 1990.

[Len02] M. Lenzerini. Data integration: A theoretical perspective. (Invited tutorial). In *21st ACM Symposium on Principles of database systems*, pages 233–246, 2002.

[LS86] J. Lambek and P. Scott. *Introduction to higher order categorical logic*. Cambridge University Press, 1986.

[LSDR07] Y. Lee, M. Sayyadian, A. Doan, and A. Rosenthal. eTuner: Tuning Schema Matching Software Using Synthetic Scenarios. *The VLDB Journal*, 16(1):97–122, 2007.

[MBHR05] S. Melnik, P. Bernstein, A. Halevy, and E. Rahm. Supporting executable mappings in model management. In *Proc. SIGMOD'05*. ACM Press, 2005.

[MBR01] J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, 2001.

[MHH00] R. J. Miller, L. M. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *Proc. VLDB*, pages 77–88, 2000.

[MIR93] R. Miller, Y. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. Very Large Data Bases*, pages 120–133, 1993.

[Mot87] A. Motro. Superviews: Virtual integration of multiple databases. *IEEE TOSE*, 13(7):785–798, 1987.

[PB03] R. Pottinger and P. Bernstein. Merging models based on given correspondences. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, 2003.

[PVM$^+$02] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *Proc. VLDB*, pages 598–609, 2002.

[RB01] E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[SCE$^+$07] Rick Salay, Marsha Chechik, Steve M. Easterbrook, Zinovy Diskin, Pete McCormick, Shiva Nejati, Mehrdad Sabetzadeh, and Petcharat Viriyakattiyaporn. An eclipse-based tool framework for software model management. In *ETX*, pages 55–59, 2007.

[SE05] M. Sabetzadeh and S. Easterbrook. An algebraic framework for merging incomplete and inconsistent views. In *13th Int.Conference on Requirement Engineering*, 2005.

[Shi81] D. Shipman. The functional data model and the data language DAPLEX. *ACM TODS*, 6(1):140–173, 1981.

[SL90] A. Sneth and C. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 1990.

[SP92] S. Spaccapietra and C. Parent. View integration: a step forward in solving structural conflicts. *IEEE Transactions on KDE*, 1992.

[SPD92] S. Spaccapietra, C. Parent, and Y. Dupont. Model-independent assertions for integration of heterogeneous schemas. *The VLDB Journal*, 1(1), 1992.

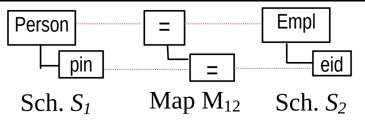
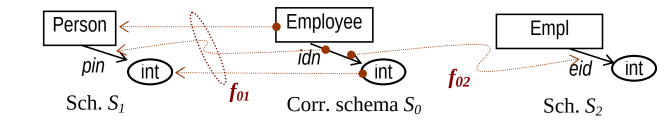
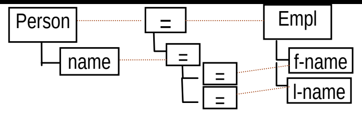
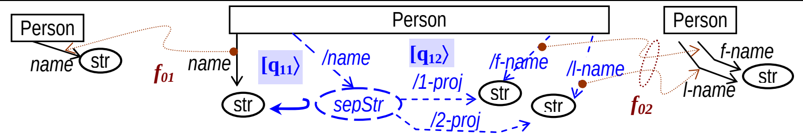
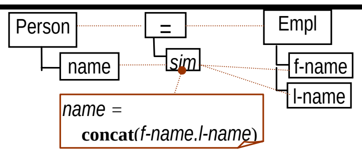
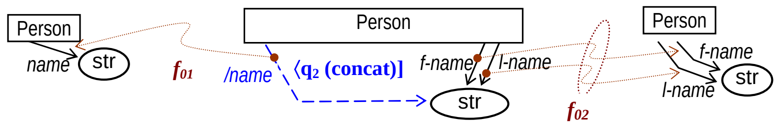
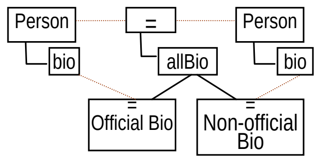
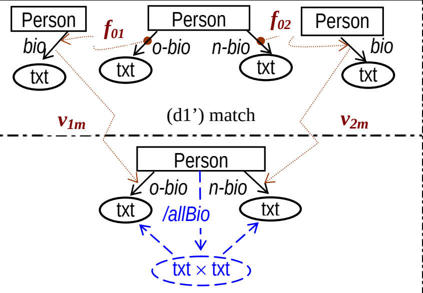
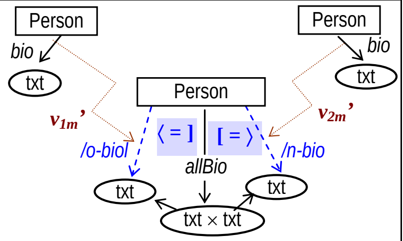
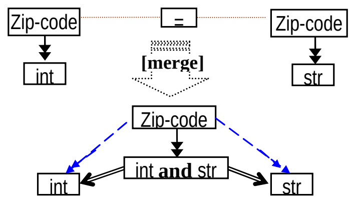
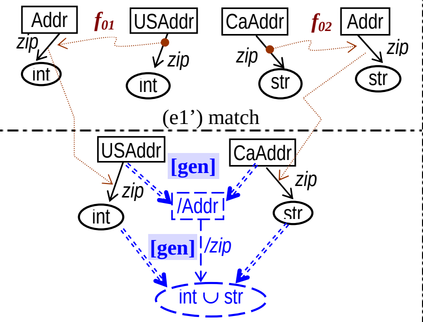
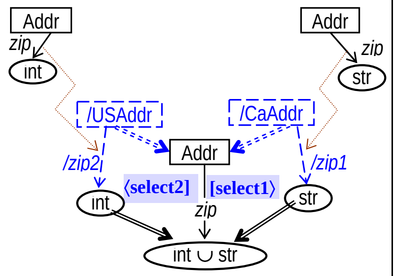
Taxonomy of conflicts in Vanilla [PB03]	… and their algebraic query-based representation (*Schema matching as Query Discovery*)	
 (a) Representation (R)-conflict of type I: resolved by equality mapping elements (EMEs)	 (a') Properties of the Corr-span: (i) Corr-head (schema S_0) does **not** contain derived elements, (ii) Corr. legs f_{01}, f_{02} are totally defined functions	
 (b) R-conflict of type II: resolved by EMEs and sub-element relationship	 (b') Properties of the Corr-span: (i) Corr-head does contain derived elements (produced by query $q_1 = q_{11} \circ q_{12}$) (ii) Corr. legs f_{01}, f_{02} are partially defined functions but (iii) For each basic element of the head, either f_{01} or f_{02} is defined.	
 (c) R-conflict of type III: resolved by SMEs and *Expression property*	 (c') The same as b'. The only difference is that a different query is used to match schemas	
 (d) R-conflict: Matching schemas with a non-mapping element allBio	 (d2') merge with some derived info added (pairing arrows *o-bio* and *n-bio*)	 (d3') restructuring the merge to a der-equivalent form
 (e) Fundamental (F)-conflict	 (e2') merge with some derived info added	 (e3') restructuring the merge to a der-equivalent form

Table 2: Conflicts via query discovery