The VLDB Journal (2008) 17:333–353
DOI 10.1007/s00778-007-0059-9

SPECIAL ISSUE PAPER

Implementing mapping composition

Philip A. Bernstein · Todd J. Green · Sergey Melnik · Alan Nash

Received: 17 February 2007 / Accepted: 10 April 2007 / Published online: 28 June 2007
© Springer-Verlag 2007

Abstract Mapping composition is a fundamental operation in metadata driven applications. Given a mapping over schemas σ_1 and σ_2 and a mapping over schemas σ_2 and σ_3, the composition problem is to compute an equivalent mapping over σ_1 and σ_3. We describe a new composition algorithm that targets practical applications. It incorporates view unfolding. It eliminates as many σ_2 symbols as possible, even if not all can be eliminated. It covers constraints expressed using arbitrary monotone relational operators and, to a lesser extent, non-monotone operators. And it introduces the new technique of left composition. We describe our implementation, explain how to extend it to support user-defined operators, and present experimental results which validate its effectiveness.

Keywords Model management · Schema mappings · Mapping composition

T.J. Green and A. Nash's work was performed during an internship at Microsoft Research.
A preliminary version of this work was published in the VLDB 2006 conference proceedings.

P. A. Bernstein (✉) · S. Melnik
Microsoft Research, Redmond, WA, USA
e-mail: philbe@microsoft.com

S. Melnik
e-mail: melnik@microsoft.com

T. J. Green
University of Pennsylvania, Philadelphia, PA, USA
e-mail: tjgreen@cis.upenn.edu

A. Nash
IBM Almaden Research Center,
San Jose, CA, USA
e-mail: anash@us.ibm.com

1 Introduction

A mapping is a relationship between the instances of two schemas. Some common types of mappings are relational queries, relational view definitions, global-and-local-as-view (GLAV) assertions, XQuery queries, and XSL transformations. The manipulation of mappings is at the core of many important data management problems, such as data integration, database design, and schema evolution. Hence, general-purpose algorithms for manipulating mappings have broad application to data management.

Data management problems like those above often require that mappings be composed. The composition of a mapping m_{12} between schemas σ_1 and σ_2 and a mapping m_{23} between schemas σ_2 and σ_3 is a mapping between σ_1 and σ_3 that captures the same relationship between σ_1 and σ_3 as m_{12} and m_{23} taken together.

Given that mapping composition is useful for a variety of database problems, it is desirable to develop a general-purpose composition component that can be reused in many application settings, as was proposed in [1,2]. This paper reports on the development of such a component, an implementation of a new algorithm for composing mappings between relational schemas. Compared to past approaches, the algorithm handles more expressive mappings, makes a best-effort when it cannot obtain a perfect answer, includes several new heuristics, and is designed to be extensible.

1.1 Applications of mapping composition

Composition arises in many practical settings. In data integration, a query needs to be composed with a view definition. If the view definition is expressed using global-as-view (GAV), then this is an example of composing two functional mappings: a view definition that maps a database to a

view, and a query that maps a view to a query result. The standard approach is view unfolding, where references to the view in the query are replaced by the view definition [9]. View unfolding is simply function composition, where a function definition (i.e., the body of the view) is substituted for references to the function (i.e., the view schema) in the query.

In peer-to-peer data management, composition is used to support queries and updates on peer databases. When two peer databases are connected through a sequence of mappings between intermediate peers, these mappings can be composed to relate the peer databases directly. In Piazza [10], such composed mappings are used to reformulate XML queries. In the ORCHESTRA collaborative data sharing system [11], updates are propagated using composed mappings to avoid materializing intermediate relations.

A third example is schema evolution, where a schema σ_1 evolves to become a schema σ_1'. The relationship between σ_1' and σ_1 can be described by a mapping. After σ_1 has evolved, any existing mappings involving σ_1, such as a mapping from σ_1 to schema σ_2, must now be upgraded to a mapping from σ_1' to σ_2. This can be done by composing the σ_1'–σ_1 mapping with the σ_1–σ_2 mapping. Depending on the application, one or both of these mappings may be non-functional, in which case composing mappings is no longer simply function composition.

A different schema evolution problem arises when an initial schema σ_1 is modified by two independent designers, producing schemas σ_2 and σ_3. To merge them into a single schema, we need a mapping between σ_2 and σ_3 that describes their overlapping content [3,8]. This σ_2–σ_3 mapping can be obtained by composing the σ_1–σ_2 and σ_1–σ_3 mappings. Even if the latter two mappings are functions, one of them needs to be inverted before they can be composed. Since the inverse of a function may not be a function, this too entails the composition of non-functional mappings.

Finally, consider a database design process that evolves a schema σ_1 via a sequence of incremental modifications. This produces a sequence of mappings between successive versions of the schema, from σ_1 to σ_2, then to σ_3, and so forth, until the desired schema σ_n is reached. At the end of this process, a mapping from σ_1 to the evolved schema σ_n is needed, for example, as input to the schema evolution scenarios above. This mapping can be obtained by composing the mappings between the successive versions of the schema. The following example illustrates this last scenario.

Example 1 Consider a schema editor, in which the designer modifies a database schema, resulting in a sequence of schemas with mappings between them. She starts with the schema σ_1

Movies(<u>mid</u>, name, year, rating, genre, theater)

where *mid* means movie identifier. The designer decides that only 5-star movies and no ‘theater’ or ‘genre’ should be present in the database; she edits the table obtaining the following schema σ_2 and mapping m_{12}:

FiveStarMovies(<u>mid</u>, name, year)

$$\pi_{\text{mid,name,year}}(\sigma_{rating=5}(\textbf{Movies})) \subseteq \textbf{FiveStarMovies}$$

To improve the organization of the data, the designer then splits the **FiveStarMovies** table into two tables, resulting in a new schema σ_3 and mapping m_{23}

Names(<u>mid</u>, name) **Years**(<u>mid</u>, year)

$$\pi_{\text{mid,name,year}}(\textbf{FiveStarMovies}) \subseteq \textbf{Names} \bowtie \textbf{Years}$$

The system composes mappings m_{12} and m_{23} into a new mapping m_{13}:

$$\pi_{\text{mid,name}}(\sigma_{rating=5}(\textbf{Movies})) \subseteq \textbf{Names}$$
$$\pi_{\text{mid,year}}(\sigma_{rating=5}(\textbf{Movies})) \subseteq \textbf{Years}$$

With this mapping, the designer can now migrate data from the old schema to the new schema, reformulate queries posed over one schema to equivalent queries over the other schema, etc.

1.2 Related work

Mapping composition is a challenging problem. Madhavan and Halevy [5] showed that the composition of two given mappings expressed as GLAV formulas may not be expressible in a finite set of first-order constraints. Fagin et al. [4] showed that the composition of certain kinds of first-order mappings may not be expressible in any first-order language, even by an infinite set of constraints. That is, that language is not *closed* under composition. Nash et al. [7] showed that for certain classes of first-order languages, it is undecidable to determine whether there is a finite set of constaints in the same language that represents the composition of two given mappings. These results are sensitive to the particular class of mappings under consideration. But in all cases the mapping languages are first-order and are therefore of practical interest.

In [4], Fagin et al. introduced a second-order mapping language that *is* closed under composition, namely second-order source-to-target tuple-generating dependencies. A second-order language is one that can quantify over function and relation symbols. A tuple-generating dependency specifies an inclusion of two conjunctive queries, $Q_1 \subseteq Q_2$. It is called source-to-target when Q_1 refers only to symbols from the source schema and Q_2 refers only to symbols from the target schema. The second-order language of [4] uses existentially quantified function symbols, which essentially can be thought of as Skolem functions. Fagin et al. present a

composition algorithm for this language and show it can have practical value for some data management problems, such as data exchange. However, using it for that purpose requires a custom implementation, since the language of second-order tuple-generating dependencies is not supported by standard SQL-based database tools.

Yu and Popa [13] considered mapping composition for second-order source-to-target constraints over nested relational schemas in support of schema evolution. They presented a composition algorithm similar to the one in [4], with extensions to handle nesting, and with signficant attention to minimizing the size of the result. They reported on a set of experiments using mappings on both synthetic and real life schemas, to demonstrate that their algorithm is fast and is effective at minimizing the size of the result.

Nash et al. [7] studied the composition of first-order constraints that are not necessarily source-to-target. They consider dependencies that can express key constraints and inclusions of conjunctive queries $Q_1 \subseteq Q_2$ where Q_1 and Q_2 may reference symbols from both the source and target schema. They do not allow existential quantifiers over function symbols. The composition of constraints in this language is not closed and determining whether a composition result exists is undecidable. Nevertheless, they gave an algorithm that produces a composition, if it halts (which it may not do).

Like Nash et al. [7], we explore the mapping composition problem for constraints that are not restricted to being source-to-target. Our algorithm strictly extends that of Nash et al. [7], which in turn strictly extends that of Fagin et al. [4] for source-to-target embedded dependencies. If the input is a set of source-to-target embedded dependencies, our algorithm behaves similarly to that in [4], except that as in [7] we also attempt to express the result as embedded dependencies through a deskolemization step. It is known from results in [4] that such a step can not always succeed. Furthermore, we also apply a "left-compose" step which allows the algorithm to handle mappings on which the algorithm in [7] fails.

1.3 Contributions

Given the inherent difficulty of the problem and limitations of past approaches, we recognized that compromises and special features would be needed to produce a mapping composition algorithm of practical value. The first issue was which language to choose.

Algebra-based rather than logic-based. We wanted our composition algorithm to be directly usable by existing database tools. We therefore chose a relational algebraic language: Each mapping is expressed as a set of constraints, each of which is either a containment or equality of two relational algebraic expressions. This language extends the algebraic dependencies of [12]. Each constraint is of the form $E_1 = E_2$ or $E_1 \subseteq E_2$ where E_1 and E_2 are arbitrary relational expressions containing not only select, project, and join but possibly many other operators. Calculus-based languages have been used in all past work on mapping composition we know of. We chose relational algebra because it is the language implemented in all relational database systems and most tools. It is therefore familiar to the developers of such systems, who are the intended users of our component. It also makes it easy to extend our language simply by allowing the addition of new operators. Notice that the relational operators we handle are sufficient to express embedded dependencies. Therefore, the class of mappings which our algorithm accepts includes embedded dependencies and, by allowing additional operators such as set difference, goes beyond them.

Eliminates one symbol at a time. Our algorithm for composing these types of algebraic mappings gives a partial solution when it is unable to find a complete one. The heart of our algorithm is a procedure to eliminate relation symbols from the intermediate signature σ_2. Such elimination can be done one symbol at a time. Our algorithm makes a best effort to eliminate as many relation symbols from σ_2 as possible, even if it cannot eliminate all of them. By contrast, if the algorithm in [7] is unable to produce a mapping over σ_1 and σ_3 with no σ_2-symbols, it simply runs forever or gives up. In some cases it may be better to eliminate some symbols from σ_2 successfully, rather than insist on either eliminating all of them or failing. Thus, the resulting mapping may be over σ_1, σ_2', and σ_3, where σ_2' is a subset of σ_2 instead of over just σ_1 and σ_3.

To see the value of this best-effort approach, consider a composition that produces a mapping m that contains a σ_2-symbol S. If m is later composed with another mapping, it is possible that the latter composition can eliminate S. (We will see examples of this later, in our experiments.) Also, the inability to eliminate S may be inherent in the given mappings. For example, S may be involved in a recursive computation that cannot be expressed purely in terms of σ_1 and σ_3 such those of Theorem 1 in [7]:

$$R \subseteq S, \quad S = \mathrm{tc}(S), \quad S \subseteq T$$

where $\sigma_1 = \{R\}$, $\sigma_2 = \{S\}$, $\sigma_3 = \{T\}$ with R, S, T binary and where the middle constraint says that S is transitively closed. In this case, S cannot be eliminated, but is definable as a recursive view on R and can be added to σ_1. To use the mapping, those non-eliminated σ_2-symbols may need to be populated as intermediate relations that will be discarded at the end. In this example this involves low computational cost. In many applications it is better to have such an approximation to a desired composition mapping than no mapping at all. Moreover, in many cases the extra cost associated with maintaining the extra σ_2 symbols is low.

Tolerance for unknown or partially known operators. Instead of rejecting an algebraic expression because it contains unknown operators which we do not know how to

handle, our algorithm simply delays handling such operators as long as possible. Sometimes, it needs no knowledge at all of the operators involved. This is the case, for example, when a subexpression that contains an unknown operator can be replaced by another expression. At other times, we need only partial knowledge about an operator. Even if we do not have the partial knowledge we need, our algorithm does not fail globally, but simply fails to eliminate one or more symbols that perhaps it could have eliminated if it had additional knowledge about the behavior of the operator.

Use of monotonicity. One type of partial knowledge that we exploit is monotonicity of operators. An operator is monotone in one of its relation symbol arguments if, when tuples are added to that relation, no tuples disappear from its output. For example, select, project, join, union, and semijoin are all monotone. Set difference (e.g., $R - S$) and left outerjoin are monotone in their first argument (R) but not in their second (S). Our key observation is that when an operator is monotone in an argument, that argument can sometimes be replaced by a expression from another constraint. For example, if we have $E_1 \subseteq R$ and $E_2 \subseteq S$, then in some cases it is valid to replace R by E_1 in $R - S$, but not to replace S by E_2. Athough existing algorithms only work with select project, join and union, this observation enables our algorithm to handle outerjoin, set difference, and anti-semijoin. Moreover, our algorithm can handle nulls and bag semantics in many cases.

Normalization and denormalization. We call left-normalization the process of bringing the constraints to a form where a relation symbol S that we are trying to eliminate appears in a single constraint alone on the left. The result is of the form $S \subseteq E$ where E is an expression. We define right-normalization similarly. Normalization may introduce "pseudo-operators" such as Skolem functions which then need to be eliminated by a denormalization step. Currently we do not do much left-normalization. Our right-normalization is more sophisticated and, in particular, can handle projections by Skolemization. The corresponding denormalization is very complex. An important observation here is that normalization and denormalization are general steps which may possibly be extended on an operator-by-operator basis.

Left compose. One way to eliminate a relation symbol S is to replace S's occurrences in some constraints by the expression on the other side of a constraint that is normalized for S. There are two versions of this replacement, right compose and left compose. In right compose, we use a constraint $E \subseteq S$ that is right-normalized for S and substitute E for S on the left side of a constraint that is monotonic in S, such as transforming $R \times S \subseteq T$ into $R \times E \subseteq T$, thereby eliminating S from the constraint. Right composition is an extension of the algorithms in [4,7]. We also introduce left compose, which handles some additional cases where right compose fails. Suppose we have the constraints $E_2 \subseteq M(S)$ and $S \subseteq E_1$, where $M(S)$ is an expression that is monotonic in S but which we either do not know how to right-normalize or which would fail to right-denormalize. Then left compose immediately yields $E_2 \subseteq M(E_1)$.

Extensibility and modularity. Our algorithm is extensible by allowing additional information to be added separately for each operator in the form of information about monotonicity and rules for normalization and denormalization. Many of the steps are rule-based and implemented in such a way that it is easy to add rules or new operators. Therefore, our algorithm can be easily adapted to handle additional operators without specialized knowledge about its overall design. Instead, all that is needed is to add new rules.

Experimental study. We implemented our algorithm and ran experiments to study its behavior. We used composition problems drawn from the recent literature [4,6,7], and a set of large synthetic composition tasks, in which the mappings were generated by composing sequences of elementary schema modifications. We used these mappings to test the scalability and effectiveness of our algorithm in a systematic fashion. Across a range of composition tasks, it eliminated 50–100% of the symbols and usually ran in under a second. We see this study as a step toward developing a benchmark for composition algorithms.

The rest of the paper is organized as follows. Section 2 presents notation needed to describe the algorithm. Section 3 describes the algorithm itself, starting with a high-level description and then drilling into the details of each step, one-by-one. Section 4 presents our experimental results. Section 5 is the conclusion.

2 Preliminaries

We adopt the unnamed perspective which references the attributes of a relation by index rather than by name. A *relational expression* is an expression formed using base relations and the *basic operators* union $\cup$, intersection $\cap$, cross product $\times$, set difference -, projection π and selection σ as follows. The name S of a relation is a relational expression. If E_1 and E_2 are relational expressions, then so are

$$E_1 \cup E_2 \qquad E_1 \cap E_2 \qquad E_1 \times E_2$$
$$E_1 - E_2 \qquad \sigma_c(E_1) \qquad \pi_I(E_1)$$

where c is an arbitrary boolean formula on attributes (identified by index) and constants and I is a list of indexes. The meaning of a relational expression is given by the standard set semantics. To simplify the presentation in this paper, we focus on these basic six relational operators and view the join operator $\bowtie$ as a derived operator formed from $\times$, π, and σ. We also allow for user-defined operators to appear in expressions. The basic operators should therefore be considered as

those which have "built-in" support, but they are not the only operators supported.

The basic operators differ in their behavior with respect to arity. Assume expression E_1 has arity r and expression E_2 has arity s. Then the arity of $E_1 \cup E_2$, $E_1 \cap E_2$, and $E_1 - E_2$ for $r = s$ is r; the arity of $E_1 \times E_2$ is $r+s$; the arity of $\sigma_c(E_1)$ is r; and the arity of $\pi_I(E_1)$ is $|I|$. We will sometimes denote the arity of an expression E by arity(E).

We define an additional operator which may be used in relational expressions called the *Skolem function*. A Skolem function has a name and a set of indexes. Let f be a Skolem function on indexes I. Then $f_I(E_1)$ is an expression of arity $r + 1$. Intuitively, the meaning of the operator is to add an attribute to the output, whose values are some function f of the attribute values identified by the indexes in I. We do not provide a formal semantics here. Skolem functions are used internally as a technical device in Sect. 3.5.

We consider constraints of two forms. A *containment constraint* is a constraint of the form $E_1 \subseteq E_2$, where E_1 and E_2 are relational expressions. An *equality constraint* is a constraint of the form $E_1 = E_2$, where E_1 and E_2 are relational expressions. We denote sets of constraints with capital Greek letters and individual constraints with lowercase Greek letters.

A *signature* is a function from a set of relation symbols to positive integers which give their arities. In this paper, we use the terms signature and schema synonymously. We denote signatures with the letter σ. (We denote relation symbols with uppercase Roman letters R, S, T, etc.) We sometimes abuse notation and use the same symbol σ to mean simply the domain of the signature (a set of relations).

An *instance* of a database schema is a database that conforms to that schema. We use uppercase Roman letters A, B, C, etc to denote instances. If A is an instance of a database schema containing the relation symbol S, we denote by S^A the contents of the relation S in A.

Given a relational expression E and a relational symbol S, we say that E is *monotone* in S if whenever instances A and B agree on all relations except S and $S^A \subseteq S^B$, then $E(A) \subseteq E(B)$. In other words, E is monotone in S if adding more tuples to S only adds more tuples to the query result. We say that E is *anti-monotone* in S if whenever A and B agree on all relations except S and $S^A \subseteq S^B$, then $E(A) \supseteq E(B)$.

The *active domain* of an instance is the set of values that appear in the instance. We allow the use of a special relational symbol D which denotes the active domain of an instance. D can be thought of as a shorthand for the relational expression $\bigcup_{i=1}^{n} \bigcup_{j=1}^{a_i} \pi_j(S_i)$ where $\sigma = \{S_1, \ldots, S_n\}$ is the signature of the database and $a_i = \text{arity}(S_i)$. We also allow the use of another special relation in expressions, the empty relation $\emptyset$.

An instance A *satisfies* a containment constraint $E_1 \subseteq E_2$ if $E_1(A) \subseteq E_2(A)$. An instance A *satisfies* an equality constraint $E_1 = E_2$ if $E_1(A) = E_2(A)$. We write $A \models \xi$ if the instance A satisfies the constraint ξ and $A \models \Sigma$ if A satisfies every constraint in Σ. Note that $A \models E_1 = E_2$ iff $A \models E_1 \subseteq E_2$ and $A \models E_2 \subseteq E_1$.

Example 2 The constraint that the first attribute of a binary relation S is the key for the relation, which can be expressed in a logic-based setting as the equality-generating dependency $S(x, y), S(x, z) \to y = z$ may be expressed in our setting as a containment constraint by making use of the active domain relation

$$\pi_{24}(\sigma_{1=3}(S^2)) \subseteq \sigma_{1=2}(D^2)$$

where S^2 is short for $S \times S$ and D^2 is short for $D \times D$.

A *mapping* is a binary relation on instances of database schemas. We reserve the letter m for mappings. Given a class of constraints $\mathcal{L}$, we associate to every expression of the form $(\sigma_1, \sigma_2, \Sigma_{12})$ the mapping

$$\{\langle A, B\rangle : (A, B) \models \Sigma_{12}\}.$$

That is, it defines which instances of two schemas correspond to each other. Here Σ_{12} is a finite subset of $\mathcal{L}$ over the signature $\sigma_1 \cup \sigma_2$, σ_1 is the *input (or source) signature*, σ_2 is the *output* (*or target*) *signature*, A is a database with signature σ_1 and B is a database with signature σ_2. We assume that σ_1 and σ_2 are disjoint. (A, B) is the database with signature $\sigma_1 \cup \sigma_2$ obtained by taking all the relations in A and B together. Its active domain is the union of the active domains of A and B. In this case, we say that m is *given by* $(\sigma_1, \sigma_2, \Sigma_{12})$.

Given two mappings m_{12} and m_{23}, the *composition* $m_{12} \circ m_{23}$ is the unique mapping

$$\{\langle A, C\rangle : \exists B \langle A, B\rangle \in m_{12} \text{ and } \langle B, C\rangle \in m_{23}\}.$$

Assume two mappings m_{12} and m_{23} are given by $(\sigma_1, \sigma_2, \Sigma_{12})$ and $(\sigma_2, \sigma_3, \Sigma_{23})$. The *mapping composition problem* is to find Σ_{13} such that $m_{12} \circ m_{23}$ is given by $(\sigma_1, \sigma_3, \Sigma_{13})$.

Given a finite set of constraints Σ over some schema σ and another finite set of constraints Σ' over some subschema σ' of σ we say that Σ is *equivalent* to Σ', denoted $\Sigma \equiv \Sigma'$, if

1. **(Soundness)** Every database A over σ satisfying Σ when restricted to only those relations in σ' yields a database A' over σ' that satisfies Σ' and
2. **(Completeness)** Every database A' over σ' satisfying the constraints Σ' can be extended to a database A over σ satisfying the constraints Σ by adding new relations in $\sigma - \sigma'$ (not limited to the domain of A').

Example 3 The set of constraints

$$\Sigma := \{R \subseteq S, \quad S \subseteq T\}$$

is equivalent to the set of constraints

$$\Sigma' := \{R \subseteq T\}.$$

Soundness. Given an instance A which satisfies Σ, we must have $R^A \subseteq S^A \subseteq T^A$ and therefore $R^A \subseteq T^A$ so if A' consists of the relations R^A and T^A, then A' satisfies Σ'.
Completeness. Given an instance A' which satisfies Σ', we must have $R^{A'} \subseteq T^{A'}$. If we make A consist of the relations $R^{A'}$ and $T^{A'}$ and we set $S^A := R^{A'}$ or $S^A := T^{A'}$, then $R^A \subseteq S^A \subseteq T^A$ and therefore A satisfies Σ.

Given this definition, we can restate the composition problem as follows. Given a set of constraints Σ_{12} over $\sigma_1 \cup \sigma_2$ and a set of constraints Σ_{23} over $\sigma_2 \cup \sigma_3$, find a set of constraints Σ_{13} over $\sigma_1 \cup \sigma_3$ such that $\Sigma_{12} \cup \Sigma_{23} \equiv \Sigma_{13}$.

3 Algorithm

3.1 Overview

At the heart of the composition algorithm (which appears at the end of this subsection), we have the procedure ELIMINATE which takes as input a finite set of constraints Σ over some schema σ that includes the relation symbol S and which produces as output another finite set of constraints Σ' over $\sigma - \{S\}$ such that $\Sigma' \equiv \Sigma$, or reports failure to do so. On success, we say that we have *eliminated* S from Σ.

Given such a procedure ELIMINATE we have several choices on how to implement COMPOSE, which takes as input three schemas σ_1, σ_2, and σ_3 and two sets of constraints Σ_{12} and Σ_{23} over $\sigma_1 \cup \sigma_2$ and $\sigma_2 \cup \sigma_3$ respectively. The goal of COMPOSE is to return a set of constraints Σ_{13} over $\sigma_1 \cup \sigma_3$. That is, its goal is to eliminate the relation symbols from σ_2. Since this may not be possible, we aim at eliminating from $\Sigma := \Sigma_{12} \cup \Sigma_{23}$ a set S of relation symbols in σ_2 which is as large as possible or which is maximal under some other criterion than the number of relation symbols in it. There are many choices about how to do this, but we do not explore them in this paper. Instead we simply follow the user-specified ordering on the relation symbols in σ_2 and try to eliminate as many as possible in that order.[1]

We therefore concentrate in the remainder of Sect. 3 on ELIMINATE. It consists of the following three steps, which we describe in more detail in following sections:

1. View unfolding
2. Left compose
3. Right compose

Each of the steps 1, 2, and 3 attempts to remove S from Σ. If any of them succeeds, ELIMINATE terminates successfully. Otherwise, ELIMINATE fails.

All three steps work in essentially the same way: given a constraint that contains S alone on one side of a constraint and an expression E on the other side, they substitute E for S in all other constraints.

Example 4 Here are three one-line examples of how each of these three steps transforms a set of two constraints into an equivalent set with just one constraint in which S has been eliminated:

1. $S = R \times T, U - S \subseteq U \Rightarrow U - (R \times T) \subseteq U$
2. $R \subseteq S \cap V, S \subseteq T \times U \Rightarrow R \subseteq (T \times U) \cap V$
3. $\pi_{21}(T) \subseteq S, S - U \subseteq R \Rightarrow \pi_{21}(T) - U \subseteq R$

In case 1, we use view unfolding to replace S with $R \times T$. In case 2, we use left compose to replace S with $T \times U$. Notice that this is correct because $S \cap V$ is monotone in S (we discuss monotonicity below). Finally, in case 3, we use right compose to replace S with $\pi_{21}(T)$. This is correct because $S - U$ is monotone in S.

To perform such a substitution we need an expression that contains S alone on one side of a constraint. This holds in the example, but is typically not the case. Another key feature of our algorithm is that it performs *normalization* as necessary to put the constraints into such a form. In the case of left and right compose, we also need all other expressions that contain S to be monotone in S.

Example 5 We can normalize the constraints

$$S - \sigma_{1=2}(U) \subseteq R, \quad \pi_{14}(T \times U) \subseteq S \cap V$$

by splitting the second one in two to obtain

$$S - \sigma_{1=2}(U) \subseteq R, \quad \pi_{14}(T \times U) \subseteq S, \quad \pi_{14}(T \times U) \subseteq V.$$

With the constraints in this form, we can eliminate S using right composition, obtaining

$$\pi_{14}(T \times U) - \sigma_{1=2}(U) \subseteq R, \quad \pi_{14}(T \times U) \subseteq V.$$

We now give a more detailed technical overview of the three steps we have just introduced. To simplify the discussion below, we take $\Sigma_0 := \Sigma$ to be the input to ELIMINATE and Σ_s to be the result after step s is complete. We use E, E_1, E_2 to stand for arbitrary relational expressions and $M(S)$ to stand for a relational expression monotonic in S.

1. **View unfolding.** We look for a constraint ξ of the form $S = E_1$ in Σ_0 where E_1 is an arbitrary expression that does not contain S. If there is no such constraint, we set $\Sigma_1 := \Sigma_0$ and go to step 2. Otherwise, to obtain Σ_1 we remove ξ and replace every occurrence of S in every

[1] Note that which symbols will be eliminated will in general depend on this user-defined order. Consider, for example, the constraints in the proof of Theorem 1 in [7] plus the additional view constraint $S_1 = S_2$: exactly one of S_1 or S_2 can be eliminated and this will depend on the order.

other constraint in Σ_0 with E_1. Then $\Sigma_1 \equiv \Sigma_0$. Soundness is obvious and to show completeness it is enough to set $S = E_1$.

2. **Left compose.** If S appears on both sides of some constraint in Σ_1, we exit. Otherwise, we convert every equality constraint $E_1 = E_2$ that contains S into two containment constraints $E_1 \subseteq E_2$ and $E_2 \subseteq E_1$ to obtain Σ_1'.
Next we check Σ_1' for *right-monotonicity* in S. That is, we check whether every expression E in which S appears to the right of a containment constraint is monotonic in S. If this check fails, we set $\Sigma_2 := \Sigma_1$ and go to step 3.
Next we left-normalize every constraint in Σ_1' for S to obtain Σ_1''. That is, we replace all constraints in which S appears on the left with a single equivalent constraint ξ of the form $S \subseteq E_1$. That is, S appears alone on the left in ξ. This is not always possible; if we fail, we set $\Sigma_2 := \Sigma_1$ and go to step 3.
If S does not appear on the left of any constraint, then we add to Σ_1'' the constraint $\xi : S \subseteq E_1$ and we set $E_1 := D^r$ where r is the arity of S. Here D is a special symbol which stands for the active domain. Clearly, any S satisfies this constraint.
Now to obtain Σ_1''' from Σ_1'' we remove ξ and for every constraint in Σ_1'' of the form $E_2 \subseteq M(S)$ where M is monotonic in S, we put a constraint of the form $E_2 \subseteq M(E_1)$ in Σ_1'''. We call this step *basic left-composition*. Finally, to the extent that our knowledge of the operators allows us, we attempt to eliminate D^r (if introduced) from any constraints, to obtain Σ_2. For example $E_1 \cap D^r$ becomes E_1.
Then $\Sigma_2 \equiv \Sigma_1$. Soundness follows from monotonicity since $E_2 \subseteq M(S) \subseteq M(E_1)$ and to show completeness it is enough to set $S := E_1$.

3. **Right compose.** Right compose is dual to left-compose. We check for left-monotonicity and we right-normalize as in the previous step to obtain Σ_2'' with a constraint ξ of the form $E_1 \subseteq S$. If S does not appear on the right of any constraint, then we add to Σ_2'' the constraint $\xi : E_1 \subseteq S$ and set $E_1 := \emptyset$. Clearly, any S satisfies this constraint.
In order to handle projection during the normalization step we may introduce Skolem functions. For example $R \subseteq \pi_1(S)$ where R is unary and S is binary becomes $f(R) \subseteq S$. The expression $f(R)$ is binary and denotes the result of applying some unknown Skolem function f to the expression R. The right-normalization step always succeeds for select, project, and join, but may fail for other operators. If we fail to normalize, we set $\Sigma_3 := \Sigma_2$ and we exit.
Now to obtain Σ_2''' from Σ_2'' we remove ξ and for every constraint in Σ_2'' of the form $M(S) \subseteq E_2$ where M is monotonic in S, we put a constraint of the form $M(E_1) \subseteq E_2$ in Σ_2'''. We call this step *basic right-composition*. Finally, to the extent that our knowledge of the operators allows us, we attempt to eliminate $\emptyset$ (if introduced) from any constraints, to obtain Σ_3. For example $E_1 \cup \emptyset$ becomes E_1.
Then $\Sigma_2''' \equiv \Sigma_2$. Soundness follows from monotonicity since $M(E_1) \subseteq M(S) \subseteq E_2$ and to show completeness it is enough to set $S := E_1$.
Since during normalization we may have introduced Skolem functions, we now need a right-denormalization step to remove such Skolem functions. Following [7], we call this part deskolemization. Deskolemization is very complex and may fail. If it does, we set $\Sigma_3 := \Sigma_2$ and we exit. Otherwise, we set Σ_3 to be the result of deskolemization.
In a more general setting, right-denormalization may take additional steps to remove auxiliary operators introduced during right-normalization. Similarly, it is possible that in the future we will have a left-denormalization step to remove auxiliary operators introduced during left-normalization. However, currently, right-denormalization consists only of deskolemization.

Procedure ELIMINATE
Input: Signature σ
Constraints Σ
Relation Symbol S
Output: Constraints Σ' over σ or $\sigma - \{S\}$

1. $\Sigma' :=$ VIEWUNFOLD(Σ, S). On success, return Σ'.
2. $\Sigma' :=$ LEFTCOMPOSE(Σ, S). On success, return Σ'.
3. $\Sigma' :=$ RIGHTCOMPOSE(Σ, S). On success, return Σ'.
4. Return Σ and indicate failure.

Procedure COMPOSE
Input: Signatures $\sigma_1, \sigma_2, \sigma_3$
Constraints Σ_{12}, Σ_{23}
Relation Symbol S
Output: Signature σ satisfying $\sigma_1 \cup \sigma_3 \subseteq \sigma \subseteq \sigma_1 \cup \sigma_2 \cup \sigma_3$
Constraints Σ over σ

1. Set $\sigma := \sigma_1 \cup \sigma_2 \cup \sigma_3$.
2. Set $\Sigma = \Sigma_{12} \cup \Sigma_{23}$.
3. For every relation symbol $S \in \sigma_2$ do:
4. $\quad \Sigma :=$ ELIMINATE(σ, Σ, S)
5. $\quad$ On success, set $\sigma := \sigma - \{S\}$.
6. Return σ, Σ.

Theorem 1 *Algorithm* COMPOSE *is correct.*

Proof To show that the algorithm COMPOSE is correct, it is enough to show that algorithm ELIMINATE is correct. That is, we must show that on input Σ, ELIMINATE returns either Σ

or Σ' from which S has been eliminated and such that Σ' is equivalent to Σ.

To show this, we show that every step preserves equivalence. View unfolding preserves equivalence since it removes the constraint $S = E_1$ and replaces every occurrence of S in the remaining constraints with E_1. Soundness is obvious and completeness follows from the fact that if Σ' is satisfied, we can set S to the value of E_1 to satisfy Σ.

The transformation steps of left-compose clearly preserve equivalence and the basic left-compose step removes the constraint $S \subseteq E_1$ and replaces every other occurrence of S in the remaining constraints with E_1. Soundness follows from monotonicity and transitivity of $\subseteq$, since every constraint of the form $E_2 \subseteq M(S)$ where M is an expression monotonic in S is replaced by $E_2 \subseteq M(E_1)$. Since $S \subseteq E_1$, monotonicity implies $M(S) \subseteq M(E_1)$ and transitivity of $\subseteq$ implies $E_2 \subseteq M(E_1)$. Completeness follows from the fact that if Σ' is satisfied, we can set S to the value of E_1 to satisfy Σ.

The soundness and completeness of right-compose are proved similarly, except that we also rely on the proof of correctness of the deskolemization algorithm in [7]. □

3.2 View unfolding

The goal of the unfold views step is to eliminate S at an early stage by applying the technique of view unfolding. It takes as input a set of constraints Σ_0 and a symbol S to be eliminated. It produces as output an equivalent set of constraints Σ_1 with S eliminated (in the success case), or returns Σ_0 (in the failure case). The step proceeds as follows. We look for a constraint ξ of the form $S = E_1$ in Σ_0 where E_1 is an arbitrary expression that does not contain S. If there is no such constraint, we set $\Sigma_1 := \Sigma_0$ and report failure. Otherwise, to obtain Σ_1 we remove ξ and replace every occurrence of S in every other constraint in Σ_0 with E_1. Note that S may occur in expressions that are not necessarily monotone in S, or that contain user-defined operators about which little is known. In either case, because S is defined by an equality constraint, the result is still an equivalent set of constraints. This is in contrast to left compose and right compose, which rely for correctness on the monotonicity of expressions in S when performing substitution.

Example 6 Suppose the input constraints are given by

$$S = R_1 \times R_2, \quad \pi_{14}(R_3 - S) \subseteq T_1, \quad T_2 \subseteq T_3 - \sigma_{2=3}(S).$$

Then unfold views deletes the first constraint and substitutes $R_1 \times R_2$ for S in the second two constraints, producing

$$\pi_{14}(R_3 - (R_1 \times R_2)) \subseteq T_1, \quad T_2 \subseteq T_3 - \sigma_{2=3}(R_1 \times R_2).$$

Note that in this example, neither left compose nor right compose would succeed in eliminating S. Left compose would fail because the expression $T_3 - \sigma_{2=3}(S)$ is not monotone in S. Right compose would fail because the expression $\pi_{14}(R_3 - S)$ is not monotone in S. Therefore view unfolding does indeed give us some extra power compared to left compose and right compose alone.

As noted above, there may not be any constraints of the form $S = E_1$. We will see below that for left and right compose, we apply some normalization rules to attempt to get to a constraint of the form $S \subseteq E_1$ or $E_1 \subseteq S$. Similarly, we could apply some transformation rules here. For example,

$$\begin{aligned} \times: \quad & E_1 \times E_2 = E_3 \;\leftrightarrow\; E_1 = \pi_I(E_3),\, E_2 = \pi_J(E_3), \\ & \qquad\qquad\qquad\quad E_3 = \pi_I(E_3) \times \pi_J(E_3) \\ \subseteq: \quad & S \subseteq E_1,\, E_1 \subseteq S \;\leftrightarrow\; S = E_1 \end{aligned}$$

where in the first rule $I = 1, \ldots, \text{arity}(E_1)$ and $J = \text{arity}(E_1) + 1, \ldots, \text{arity}(E_1) + \text{arity}(E_2)$.

However, we do not know of any rules for the other relational operators: $\cup, \cap, -, \sigma, \pi$. Therefore we do not discuss a normalization step for view unfolding (cf. Sects. 3.4.1 and 3.5.1).

3.3 Checking monotonicity

The correctness of performing substitution to eliminate a symbol S in the left compose and right compose steps depends upon the left-hand side (lhs) or right-hand side (rhs) of all constraints being monotone in S. We describe here a sound but incomplete procedure MONOTONE for checking this property. MONOTONE takes as input an expression E and a symbol S. It returns m if the expression is monotone in S, a if the expression is anti-monotone in S, i if the expression is independent of S (for example, because it does not contain S), and u (unknown) if it cannot say how the expression depends on S. For example, given the expression $S \times T$ and symbol S as input, MONOTONE returns m, while given the expression $\sigma_{c_1}(S) - \sigma_{c_2}(S)$ and the symbol S, MONOTONE returns u.

The procedure is defined recursively in terms of the six basic relational operators. In the base case, the expression is a single relational symbol, in which case MONOTONE returns m if that symbol is S, and i otherwise. Otherwise, in the recursive case, MONOTONE first calls itself recursively on the operands of the top-level operator, then performs a simple table lookup based on the return values and the operator. For the unary expressions $\sigma(E_1)$ and $\pi(E_1)$, we have that MONOTONE$(\sigma(E_1), S)$ = MONOTONE$(\pi(E_1), S)$ = MONOTONE(E_1, S) (in other words, σ and π do not affect the monotonicity of the expression). Otherwise, for the binary expressions $E_1 \cup E_2$, $E_1 \cap E_2$, $E_1 \times E_2$, and $E_1 - E_2$, there are sixteen cases to consider, corresponding to the possible values of MONOTONE(E_1, S) and MONOTONE(E_2, S).

Table 1 Recursive definition of MONOTONE for the basic binary relational operators. In the top row, E_1 abbreviates MONOTONE(E_1, S), E_2 abbreviates MONOTONE(E_2, S), etc.

E_1	E_2	$E_1 \cup E_2$, $E_1 \cap E_2$, $E_1 \times E_2$	$E_1 - E_2$
m	m	m	u
m	i	m	m
m	a	u	m
i	m	m	a
i	i	i	i
i	a	a	m
a	m	u	a
a	i	a	a
a	a	a	u
u	any	u	u
any	u	u	u

We give a couple of quick examples. We refer the reader to Table 1 for a detailed listing of the cases.

Example 7

$$\text{MONOTONE}(E_1, S) = m, \quad \text{MONOTONE}(E_2, S) = a$$
$$\Rightarrow \text{MONOTONE}(E_1 \times E_2, S) = u.$$
$$\text{MONOTONE}(E_1, S) = i, \quad \text{MONOTONE}(E_2, S) = a$$
$$\Rightarrow \text{MONOTONE}(E_1 - E_2, S) = m.$$

Note that $\times$, $\cap$, and $\cup$ all behave in the same way from the point of view of MONOTONE, that is, MONOTONE$(E_1 \cup E_2, S)$ = MONOTONE$(E_1 \cap E_2, S)$ = MONOTONE$(E_1 \times E_2, S)$, for all E_1, E_2. Set difference –, on the other hand, behaves differently than the others.

In order to support user-defined operators in MONOTONE, we just need to know the rules regarding the monotonicity of the operator in S, given the monotonicity of its operands in S. Once these rules have been added to the appropriate tables, MONOTONE supports the user-defined operator automatically.

3.4 Left compose

Recall from Sect. 3.1 that left compose consists of four main steps, once equality constraints have been converted to containment constraints. The first is to check the constraints for right-monotonicity in S, that is, to check whether every expression E in which S appears to the right of a containment constraint is monotonic in S. Section 3.3 already described the procedure for checking this. The other three steps are left normalize, basic left compose, and eliminate domain relation. In this section we describe those steps in more detail, and we give some examples to illustrate their operation.

3.4.1 Left normalize

The goal of left normalize is to put the set of input constraints in a form such that the symbol S to be eliminated appears on the left of exactly one constraint, which is of the form $S \subseteq E_2$. We say that the constraints are then in *left normal form.* In contrast to right normalize, left normalize does not always succeed even on the basic relational operators. Nevertheless, left composition is useful because it may succeed in cases where right composition fails for other reasons. We give an example of this in Sect. 3.4.2.

We make use of the following identities for containment constraints in left normalize:

$$\cup: \ E_1 \cup E_2 \subseteq E_3 \leftrightarrow E_1 \subseteq E_3, E_2 \subseteq E_3$$
$$\cap: \ E_1 \cap E_2 \subseteq E_3 \leftrightarrow E_1 \subseteq E_3 \cup (D^r - E_2)$$
$$-: \ E_1 - E_2 \subseteq E_3 \leftrightarrow E_1 \subseteq E_2 \cup E_3$$
$$\pi: \ \pi_I(E_1) \subseteq E_2 \leftrightarrow E_1 \subseteq \pi_J(E_2 \times D^s)$$
$$\sigma: \ \sigma_c(E_1) \subseteq E_2 \leftrightarrow E_1 \subseteq E_2 \cup (D^r - \sigma_c(D^r))$$

In the identities for $\cap$ and σ, r stands for arity(E_2). In the identity for π, s stands for arity(E_1) − arity(E_2) and J is defined as follows: suppose $I = i_1, \ldots, i_m$ and let $i_{m+1}, \ldots, i_n$ be the indexes of E_1 not in I, n = arity(E_1); then define $J := j_1, \ldots, j_n$ where $j_{i_k} := k$ for $1 \leq k \leq n$.

To each identity in the list, we associate a rewriting rule that takes a constraint of the form given by the lhs of the identity and produces an equivalent constraint or set of constraints of the form given by the rhs of the identity. For example, from the identity for σ we obtain a rule that matches a constraint of the form $\sigma_c(E_1) \subseteq E_2$ and rewrites it into equivalent constraints of the form $E_1 \subseteq E_2 \cup (D^r - \sigma_c(D^r))$. Note that there is at most one rule for each operator. So to find the rule that matches a particular expression, we need only look up the rule corresponding to the topmost operator in the expression.

We can assume that S is in E_1 except in the case of set difference. In the case of set difference if S is in E_2 we can still apply the rule which just removes S from the lhs.

Of the basic relational operators, the only one which may cause left normalize to fail is cross product, for which we do not know of an identity. One might be tempted to think that the constraint $E_1 \times E_2 \subseteq E_3$ could be rewritten as $E_1 \subseteq \pi_I(E_3), E_2 \subseteq \pi_J(E_3)$, where $I = 1, \ldots,$ arity(E_1) and J = arity$(E_1) + 1, \ldots,$ arity(E_1) + arity(E_2). However, the following counterexample shows that this rewriting is invalid:

Example 8 Let R, S be unary relations and let T be a binary relation. Define the instance A to be $R^A := \{1, 2\}$,

$S^A := \{1, 2\}$, $T^A := \{11, 22\}$. Then $A \models \{R \subseteq \pi_1(T), S \subseteq \pi_2(T)\}$, but $A \not\models \{R \times S \subseteq T\}$.

In addition to the basic relational operators, left normalize may be extended to handle user-defined operators by specifying a user-defined rewriting rule for each such operator.

Left normalize proceeds as follows. Let Σ_1 be the set of input constraints, and let S be the symbol to be eliminated from Σ_1. Left normalize computes a set Σ_1' of constraints as follows. Set $\Gamma_1 := \Sigma_1$. We loop as follows, beginning at $i = 1$. In the ith iteration, there are two cases:

1. If there is no constraint in Γ_i that contains S on the lhs in a complex expression, set Σ_1' to be Γ_i with all the constraints containing S on the lhs collapsed into a single constraint, which has an intersection of expressions on the right. For example, $S \subseteq E_1, S \subseteq E_2$ becomes $S \subseteq E_1 \cap E_2$. If S does not appear on the lhs of any expression, we add to Σ_1' the constraint $S \subseteq D^r$ where r is the arity of S. Finally, return success.
2. Otherwise, choose some constraint $\xi := E_1 \subseteq E_2$, where E_1 contains S. If there is no rewriting rule for the top-level operator in E_1, set $\Sigma_1' := \Sigma_1$ and return failure. Otherwise, set Γ_{i+1} to be the set of constraints obtained from Γ_i by replacing ξ with its rewriting, and iterate.

Example 9 Suppose the input constraints are given by

$$R - S \subseteq T, \quad \pi_2(S) \subseteq U.$$

where S is the symbol to be eliminated. Then left normalization succeeds and returns the constraints

$$R \subseteq S \cup T, \quad S \subseteq \pi_{21}(U \times D).$$

Example 10 Suppose the input constraints are given by

$$R \times S \subseteq T, \quad \pi_2(S) \subseteq U.$$

Then left normalization fails for the first constraint, because there is no rule for cross product.

Example 11 Suppose the input constraints are given by

$$R \times T \subseteq S, \quad U \subseteq \pi_2(S).$$

Since there is no constraint containing S on the left, left normalize adds the trivial constraint $S \subseteq D^2$, producing

$$R \times T \subseteq S, \quad U \subseteq \pi_2(S), \quad S \subseteq D^2.$$

3.4.2 Basic left compose

Among the constraints produced by left normalize, there is a single constraint $\xi := S \subseteq E_1$ that has S on its lhs. In basic left compose, we remove ξ from the set of constraints, and we replace every other constraint of the form $E_2 \subseteq M(S)$, where $M(S)$ is monotonic in S, with a constraint of the form $E_2 \subseteq M(E_1)$. This is easier to understand with the help of a few examples.

Example 12 Consider the constraints from Example 9 after left normalization:

$$R \subseteq S \cup T, \quad S \subseteq \pi_{21}(U \times D).$$

The expression $S \cup T$ is monotone in S. Therefore, we are able to left compose to obtain

$$R \subseteq \pi_{21}(U \times D) \cup T.$$

Note although the input constraints from Example 9 could just as well be put in right normal form (by adding the trivial constraint $\emptyset \subseteq S$), right compose would fail, because the expression $R - S$ is not monotone in S. Thus left compose does indeed give us some additional power compared to right compose.

Example 13 We continue with the constraints from Example 11:

$$R \times T \subseteq S, \quad U \subseteq \pi_2(S), \quad S \subseteq D^2.$$

We left compose and obtain

$$R \times T \subseteq D^2, \quad U \subseteq \pi_2(D^2).$$

Note that the active domain relation D occurs in these constraints. In the next section, we explain how to eliminate it.

3.4.3 Eliminate domain relation

We have seen that left compose may produce a set of constraints containing the symbol D which represents the active domain relation. The goal of this step is to eliminate D from the constraints, to the extent that our knowledge of the operators allows, which may result in entire constraints disappearing in the process as well. We use rewriting rules derived from the following identities for the basic relational operators:

$$E_1 \cup D^r = D^r \qquad E_1 \cap D^r = E_1$$
$$E_1 - D^r = \emptyset \qquad \pi_I(D^r) = D^{|I|}$$

(We do not know of any identities applicable to cross product or selection.) In addition, the user may supply rewriting rules for user-defined operators, which we will make use of if present. The constraints are rewritten using these rules until no rule applies. At this point, D may appear alone on the rhs of some constraints. We simply delete these, since a constraint of this form is satisfied by any instance. Note that we do not always succeed in eliminating D from the constraints. However, this is acceptable, since a constraint containing D can still be checked.

Example 14 We continue with the constraints from Examples 11 and 13:

$$R \times T \subseteq D^2, \quad U \subseteq \pi_2(D^2).$$

First, the domain relation rewriting rules are applied, yielding

$$R \times T \subseteq D^2, \quad U \subseteq D,$$

Then, since both of these constraints have the domain relation alone on the rhs, we are able to simply delete them.

3.5 Right compose

Recall from Sect. 3.1 that right compose proceeds through five main steps. The first step is to check that every expression E that appears to the left of a containment constraint is monotonic in S. The procedure for checking this was described in Sect. 3.3. The other four steps are right normalize, basic right compose, right-denormalize, and eliminate empty relations. In this section, we describe these steps in more detail and provide some examples.

3.5.1 Right normalize

Right normalize is dual to left normalize. The goal of right normalize is to put the constraints in a form where S appears on the rhs of exactly one constraint, which has the form $E_1 \subseteq S$. We say that the constraints are then in *right normal form*. We make use of the following identities for containment constraints in right normalization:

$$\begin{aligned}\cup:\ & E_1 \subseteq E_2 \cup E_3 \leftrightarrow E_1 - E_3 \subseteq E_2\\ & \leftrightarrow E_1 - E_2 \subseteq E_3\end{aligned}$$

$$\cap:\ E_1 \subseteq E_2 \cap E_3 \leftrightarrow E_1 \subseteq E_2, E_1 \subseteq E_3$$

$$\times:\ E_1 \subseteq E_2 \times E_3 \leftrightarrow \pi_I(E_1) \subseteq E_2, \pi_J(E_1) \subseteq E_2$$

$$-:\ E_1 \subseteq E_2 - E_3 \leftrightarrow E_1 \subseteq E_2, E_1 \cap E_3 \subseteq \emptyset$$

$$\begin{aligned}\pi:\ & E_1 \subseteq \pi_I(E_2) \leftrightarrow f_J(E_1) \subseteq \pi_{I'}(E_2)\\ & \leftrightarrow \pi_J(E_1) \subseteq E_2\end{aligned}$$

$$\sigma:\ E_1 \subseteq \sigma_c(E_2) \leftrightarrow E_1 \subseteq E_2, E_1 \subseteq \sigma_c(D^r)$$

In the identity for $\times$, $I := 1, \ldots, \text{arity}(E_2)$ and $J := 1, \ldots, \text{arity}(E_3)$. The first identity for π holds if $|I| < \text{arity}(E_2)$; J is defined $J := 1, \ldots, \text{arity}(E_1)$ and I' is obtained from I by appending the first index in E_2 which does not appear in I. The second identity for π holds if $|I| = \text{arity}(E_2)$; if $I = i_1, \ldots, i_n$ then J is defined $J := j_1, \ldots, j_n$ where $j_{i_k} := k$. Finally, in the identity for σ, r stands for $\text{arity}(E_2)$.

As in left normalize, to each identity in the list, we associate a rewriting rule that takes a constraint of the form given by the lhs of the identity and produces an equivalent constraint or set of constraints of the form given by the rhs of the identity. For example, from the identity for σ we obtain a rule that matches constraints of the form $E_1 \subseteq \sigma_c(E_2)$ and produces the equivalent pair of constraints $E_1 \subseteq E_2$ and $E_1 \subseteq \sigma_c(D^r)$. As with left normalize, there is at most one rule for each operator. So to find the rule that matches a particular expression, we need only look up the rule corresponding to the topmost operator in the expression. In contrast to the rules used by left normalize, there is a rule in this list for each of the six basic relational operators. Therefore right normalize always succeeds when applied to constraints that use only basic relational expressions.

Just as with left normalize, user-defined operators can be supported via user-specified rewriting rules. If there is a user-defined operator that does not have a rewriting rule, then right normalize may fail in some cases.

Note that the rewriting rule for the projection operator π may introduce Skolem functions. The deskolemize step will later attempt to eliminate any Skolem functions introduced by this rule. If we have additional knowledge about key constraints for the base relations, we use this to minimize the list of attributes on which the Skolem function depends. This increases our chances of success in deskolemize.

Example 15 Given the constraint

$$\pi_{24}(\sigma_{1=3}(S \times S)) \subseteq \sigma_{1=2}(D \times D)$$

which says that the first attribute of S is a key (cf. Example 2) and

$$f_{12}(S) \subseteq \pi_{142}(\sigma_{2=3}(R \times R))$$

which says that for every edge in S, there is a path of length 2 in R, we can reduce the attributes on which f depends in the second constraint to just the first one. That is, we can replace f_{12} with f_1.

Right normalize proceeds as follows. Let Σ_2 be the set of input constraints, and let S be the symbol to be eliminated from Σ_2. Right normalize computes a set Σ_2' of constraints as follows. Set $\Gamma_1 := \Sigma_2$. We loop as follows, beginning at $i = 1$. In the ith iteration, there are two cases:

1. If there is no constraint in Γ_i that contains S on the rhs in a complex expression, set Σ_2' to be the same as Γ_i but with all the constraints containing S on the rhs collapsed into a single constraint containing a union of expressions on the left. For example, $E_1 \subseteq S$, $E_2 \subseteq S$ becomes $E_1 \cup E_2 \subseteq S$. If S does not appear on the rhs of any expression, we add to Σ_2' the constraint $\emptyset \subseteq S$. Finally, return success.

2. Otherwise, choose some constraint $\xi := E_1 \subseteq E_2$, where E_2 contains S. If there is no rewriting rule corresponding to the top-level operator in E_2, set $\Sigma_2' := \Sigma_2$ and return failure. Otherwise, set Γ_{i+1} to be the set of constraints obtained from Γ_i by replacing ξ with its rewriting, and iterate.

Example 16 Consider the constraints given by

$$S \times T \subseteq U, \quad T \subseteq \sigma_{1=2}(S) \times \pi_{21}(R).$$

Right normalize leaves the first constraint alone and rewrites the second constraint, producing

$$S \times T \subseteq U, \qquad \pi_{12}(T) \subseteq S,$$
$$\pi_{12}(T) \subseteq \sigma_{1=2}(D^2), \qquad \pi_{34}(T) \subseteq \pi_{21}(R).$$

Notice that rewriting stopped for the constraint $\pi_{34}(T) \subseteq \pi_{21}(R)$ immediately after it was produced, because S does not appear on its rhs.

Example 17 Consider the constraints given by

$$R \subseteq \pi_1(S) \times \pi_2(T \cap U), \quad S \subseteq \sigma_{1=2}(T).$$

Right normalize rewrites the first constraint and leaves the second constraint alone, producing

$$f_1(\pi_1(R)) \subseteq S, \quad \pi_2(R) \subseteq T \cap U, \quad S \subseteq \sigma_{1=2}(T).$$

Note that a Skolem function f was introduced in order to handle the projection operator. After right compose, the deskolemize procedure will attempt to get rid of the Skolem function f.

3.5.2 Basic right compose

After right normalize, there is a single constraint $\xi := E_1 \subseteq S$ which has S on its rhs. In basic right compose, we remove ξ from the set of constraints, and we replace every other constraint of the form $M(S) \subseteq E_2$, where $M(S)$ is monotonic in S, with a constraint of the form $M(E_1) \subseteq E_2$. This is easier to understand with the help of a few examples.

Example 18 Recall the constraints produced by right normalize in Example 16:

$$S \times T \subseteq U, \qquad \pi_{12}(T) \subseteq S,$$
$$\pi_{12}(T) \subseteq \sigma_{1=2}(D^2), \qquad \pi_{34}(T) \subseteq \pi_{21}(R).$$

Given those constraints as input, basic right compose produces

$$\pi_{12}(T) \times T \subseteq U, \quad \pi_{12}(T) \subseteq \sigma_{1=2}(D^2), \quad \pi_{34}(T) \subseteq \pi_{21}(R).$$

Since the constraints contain no Skolem functions, in this case we are done.

Example 19 Recall the constraints produced by right normalize in Example 17:

$$f_1(\pi_1(R)) \subseteq S, \quad \pi_2(R) \subseteq T \cap U, \quad S \subseteq \sigma_{1=2}(T).$$

Given those constraints as input, basic right compose produces

$$\pi_2(R) \subseteq T \cap U, \quad f_1(\pi_1(R)) \subseteq \sigma_{1=2}(T).$$

Note that composition is not yet complete in this case. We will need to try to complete the process by deskolemizing the constraints to get rid of f. This process is described in the next section.

3.5.3 Right-denormalize

During right-normalization, we may introduce Skolem functions in order to handle projection. For example, we transform $R \subseteq \pi_1(S)$ where R is unary and S is binary to $f_1(R) \subseteq S$. The subscript 1 indicates that f depends on position 1 of R. That is, $f_1(R)$ is a binary expression where to every value in R another value is associated by f. Thus, after basic right-composition, we may have constraints with Skolem functions in them. The semantics of such constraints is that they hold iff there exist *some* values for the Skolem functions which satisfy the constraints. The objective of the deskolemization step is to remove such Skolem functions. It is a complex 12-step procedure based on a similar procedure presented in [7].

Procedure DESKOLEMIZE(Σ)

1. Unnest
2. Check for cycles
3. Check for repeated function symbols
4. Align variables
5. Eliminate restricting atoms
6. Eliminate restricted constraints
7. Check for remaining restricted constraints
8. Check for dependencies
9. Combine dependencies
10. Remove redundant constraints
11. Replace functions with ∃-variables
12. Eliminate unnecessary ∃-variables

Here we only highlight some aspects specific to this implementation. First of all, as we already said, we use an algebra-based representation instead of a logic-based representation. A Skolem function for us is a relational operator which takes an r-ary expression and produces an expression of arity $r+1$. Our goal at the end of step 3 is to produce expressions of the form

$$\pi \sigma f g \ldots \sigma(R_1 \times R_2 \times \cdots \times R_k).$$

Here

- π selects which positions will be in the final expression,
- the outer σ selects some rows based on values in the Skolem functions,
- $f, g, \ldots$ is a sequence of Skolem functions,
- the inner σ selects some rows independently of the values in the Skolem functions, and
- $(R_1 \times R_2 \times \cdots \times R_k)$ is a cross product of possibly repeated base relations.

The goal of step 4 is to make sure that across all constraints, all these expressions have the same arity for the part after the Skolem functions. This is achieved by possibly padding with the D symbol. Furthermore, step 4 aligns the Skolem functions in such a way that across all constraints the same Skolem functions appear, in the same sequence. For example, if we have the two expressions $f_1(R)$ and $g_1(S)$ with R, S unary, step 4 rewrites them as

$$\pi_{13} g_2 f_1(R \times S) \quad \text{and} \quad \pi_{24} g_2 f_1(R \times S).$$

Here $R \times S$ is a binary expression with R in position 1 and S in position 2 and $g_2 f_1(R \times S)$ is an expression of arity 4 with R in position 1, S in position 2, f in position 3 depending only on position 1, and g in position 4 depending only on position 2.

The goal of step 5 is to eliminate the outer selection σ and of step 6 to eliminate constraints having such an outer selection. The remaining steps correspond closely to the logic-based approach.

Deskolemization is complex and may fail at several of the steps above. The following two examples illustrate some cases where deskolemization fails.

Example 20 Consider the following constraints from [4] where E, F, C, D are binary and $\sigma_2 = \{F, C\}$:

$$E \subseteq F, \quad \pi_1(E) \subseteq \pi_1(C), \quad \pi_2(E) \subseteq \pi_1(C)$$
$$\pi_{46}\, \sigma_{1=3,2=5}(F \times C \times C) \subseteq D$$

Right-composition succeeds at eliminating F to get

$$\pi_1(E) \subseteq \pi_1(C), \quad \pi_2(E) \subseteq \pi_1(C)$$
$$\pi_{46}\, \sigma_{1=3,2=5}(E \times C \times C) \subseteq D$$

Right-normalization for C yields

$$\pi_{13} f_{12}(E) \subseteq C, \quad \pi_{23} g_{12}(E) \subseteq C$$
$$\pi_{46} \sigma_{1=3,2=5}(E \times C \times C) \subseteq D$$

and basic right-composition yields 4 constraints including

$$\pi_{46} \sigma_{1=3,2=5}(E \times (\pi_{13} f_{12}(E)) \times (\pi_{13} f_{12}(E))) \subseteq D$$

which causes deskolemize to fail at step 3. Therefore, right-compose fails to eliminate C. As shown in [4] eliminating C is impossible by any means.

Example 21 Consider the following constraints where A, B are unary and $\sigma_2 = \{F, G\}$:

$$A \subseteq \pi_1(F), \quad B \subseteq \pi_1(G), \quad F \times G \subseteq T.$$

Right-normalization for F yields

$$f_1(A) \subseteq F, \quad B \subseteq \pi_1(G), \quad F \times G \subseteq T.$$

and basic right-composition yields:

$$f_1(A) \times G \subseteq T$$

which after step 4 becomes

$$\pi_{1423} f_1(A \times G) \subseteq T.$$

which causes deskolemize to fail at step 8. Therefore, right-compose fails to eliminate F. Similarly, right-compose fails to eliminate G.

3.5.4 Eliminate empty relation

Right compose may introduce the empty relation symbol $\emptyset$ into the constraints during composition in the case where S does not appear on the rhs of any constraint. In this step, we attempt to eliminate the symbol, to the extent that our knowledge of the operators allows. This is usually possible and often results in constraints disappearing entirely, as we shall see. For the basic relational operators, we make use of rewriting rules derived from the following identities for the basic relational operators:

$$E_1 \cup \emptyset = E_1 \qquad E_1 \cap \emptyset = \emptyset \qquad E_1 - \emptyset = E_1$$
$$\emptyset - E_1 = \emptyset \qquad \sigma_c(\emptyset) = \emptyset \qquad \pi_I(\emptyset) = \emptyset$$

In addition we allow the user to supply rewriting rules for user-defined operators. The constraints are rewritten using these rules until no rule applies. At this point, some constraints may have the form $\emptyset \subseteq E_2$. These constraints are then simply deleted, since they are satisfied by any instance. We do not always succeed in eliminating the empty relation symbol from the constraints. However, this is acceptable, since a constraint containing $\emptyset$ can still be checked.

3.6 Additional rules

Additional transformation rules can be used at certain steps of the algorithm for several purposes, including:

1. to eliminate obstructions to left and right compose
2. and to improve the appearance of the output constraints.

We illustrate these purposes with some examples.

Example 22 Consider the constraint

$$S \subseteq S \cap T$$

which causes both left and right compose to fail (because S appears on both sides of the constraint). It is equivalent to

$$S \subseteq T.$$

Similarly, consider the constraint $S \subseteq S \cup T$. It is equivalent to $S \subseteq S$ which can simply be deleted.

Example 23 Consider the constraint

$$R - S \subseteq T$$

in the context of right compose. $R - S$ is not monotone in S, so right compose cannot be applied to eliminate S. However, the constraint can be rewritten as

$$R - T \subseteq S$$

to allow right compose to proceed.

Example 24 Consider the constraints

$$\pi_{12}(T) \subseteq \sigma_{1=2}(D^2), \quad \pi_{12}(T) \subseteq \pi_{21}(R).$$

They can be replaced by the single constraint

$$\pi_{12}(T) \subseteq \sigma_{1=2}(\pi_{21}(R)).$$

3.7 Representing constraints

The output of our algorithm may be exponential in the size of the input, as the following example illustrates.[2]

Example 25 Consider the constraints

$$R \subseteq S_1 \circ S_1$$
$$S_1 \subseteq S_2 \circ S_2,\ S_2 \subseteq S_3 \circ S_3,\ \ldots,\ S_{n-1} \subseteq S_n \circ S_n,$$
$$S_n \subseteq T$$

where $S \circ S$ denotes the relational composition query, which can be written $\pi_{14}(\sigma_{2=3}(S \times S))$. Composing the mappings to eliminate $S_1, \ldots, S_n$ we obtain a single constraint

$$R \subseteq \underbrace{T \circ T \circ \cdots \circ T}_{2^n \text{ times}}$$

whose length is exponential in n.

The running time of the algorithm is highly sensitive to the data structures used internally to represent constraints. In a naive implementation, we represent constraints as parse trees for the corresponding strings, with no attempt to keep the representation compact by, for example, exploiting common subexpressions. With this naive tree representation, there are many cases where the running time of the algorithm is exponential in the size of the input, even when the output is not.

[2] Note that Example 5 of [7] shows a case in another setting where the number of constraints increases exponentially after composition. However, the blow-up does not occur for the same example in our setting because of our use of the union operator (which corresponds to allowing disjunctions in constraints in the logical setting of [7]).

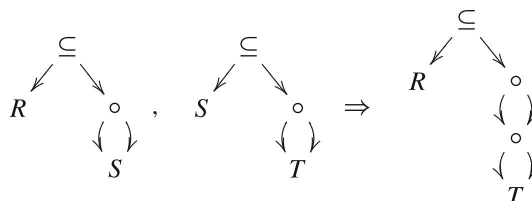
Example 26 Modify Example 25 above by removing the constraint $S_n \subseteq T$. After eliminating $S_1, \ldots, S_{n-1}$ we have the single constraint

$$R \subseteq \underbrace{S_n \circ S_n \circ \cdots \circ S_n}_{2^n \text{ times}}$$

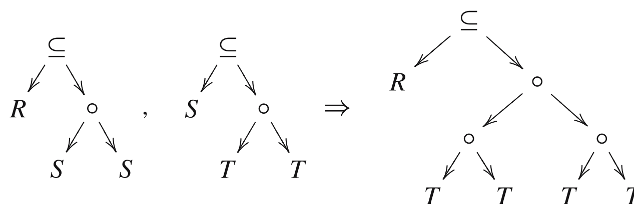
The length of this constraint is 2^n so we must have taken at least order 2^n steps to record its naive tree representation in memory. Next we eliminate S_n (by replacing it with the domain relation). After simplifications, the output is empty.

Examples such as this one motivate a search for a more compact internal representation for constraints. In our implementation, we explored using a data structure based on directed acyclic graphs (DAGs) rather than parse trees. In the DAG-based representation common subexpressions are represented only once and the substitutions in left and right compose are implemented by simply moving pointers, rather than actually replacing subtrees. We illustrate with an example.

Example 27 Consider the mappings $R \subseteq S \circ S, \quad S \subseteq T \circ T$. Their composition is $R \subseteq T \circ T \circ T \circ T$. In the DAG-based representation, these three constraints correspond to the graphs



For comparison, the corresponding trees in the naive representation are



A DAG-based representation allows us to postpone expanding the constraints to a tree representation as long as possible in order to handle cases like Example 26 efficiently.

However, the DAG-based representation also carries the practical disadvantage of increased code complexity in the normalization and de-normalization subroutines. The difficulty is that these routines must take care when rewriting expressions to avoid introducing inadvertant side-effects due to sharing of subexpressions in the DAG.

Additionally, even with the DAG-based representation, there is still a step in the algorithm, right de-normalization, which may introduce an explosion in the size of the constraints. This is due to the fact that the right-hand side of constraints must be put in a certain form not containing any union operators (see Sect. 3.5.3). Here is an example.

Example 28 Given as input the constraint

$$R \subseteq (S_1 \cup S_2) \times (S_3 \cup S_4) \times \cdots \times (S_{2n-1} \cup S_{2n})$$

right-denormalize produces 2^n constraints:

$$\begin{aligned} &R \subseteq S_1 \times S_3 \times \cdots \times S_{2n-1}, \quad &R \subseteq S_2 \times S_3 \times \cdots \times S_{2n-1}, \\ &R \subseteq S_1 \times S_4 \times \cdots \times S_{2n-1}, \quad &R \subseteq S_2 \times S_4 \times \cdots \times S_{2n-1}, \\ &\qquad \ldots, \quad &R \subseteq S_2 \times S_4 \times \cdots \times S_{2n}. \end{aligned}$$

Based on the above factors, we decided to simplify the design of our implementation by adopting a hybrid approach to constraint representation: the subtitution steps in view unfolding, left compose, and right compose use a DAG-based representation, which is expanded as needed to a naive tree representation for use in the normalization and denormalization steps. The next section gives evidence that this hybrid approach performs adequately on practical workloads.

4 Experiments

We conducted an experimental study to determine the success rate of our algorithm in eliminating symbols for various composition tasks, to measure its execution time, and to investigate the contributions of the main steps of the algorithm on its overall performance. Our study is based on the schema evolution scenarios outlined in the introduction. Specifically, we focus on schema editing and schema reconciliation tasks since mapping adaptation can be viewed as a special case of schema reconciliation where one of the input mappings is fixed.

Prior work [4,7] showed that characterizing the class of mappings that can be composed is very difficult, even when no outer joins, unions, or difference operators are present in the mapping constraints. In this section we make a first attempt to systematically explore the boundaries of mappings that can be composed. Ultimately, our goal is to develop a mapping composition benchmark that can be used to compare competing implementations of mapping composition. The experiments that we report on could form a part of such a benchmark.

4.1 Experimental setup

All composition problems used in our experiments are available for online download in a machine-readable format.[3] We designed a plain-text syntax for specifying mapping composition tasks. Mapping constraints are encoded according to the index-based algebraic notation introduced in Sect. 2. We built a parser that takes as input a textual specification of a composition problem and converts it into an internal algebraic representation, which is fed to our algorithm.

Below we show a sample run of the algorithm on mappings that contain relational operators select, project, join, cross-product, union, difference, and left outerjoin. The composition problem is stated as that of eliminating a given set of relation symbols. The output lists the symbols that the algorithm managed to eliminate and the resulting mapping constraints. This example exploits most of the techniques presented in earlier sections.

```
SCHEMA
  R1(2),R2(2),R3(2),R4(2),R5(2),
  S1(2),S2(2),S3(2),S4(2),S5(2),S6(2),
    S7(2),S8(2),
  T1(2),T2(2),T3(2),T4(2),T5(2),T6(2)
CONSTRAINTS
  S1 = P_{0,2} LEFTOUTERJOIN_{0,1:1,2} (R2 R3),
  JOIN_{0,1:1,0} (R2 R2) <= S5,
  JOIN_{0,1:0,0} (R2 R2) <= S5,
  R5 <= S7,
  P_{0} R5 <= P_{0} S8,
  P_{1} R5 <= P_{1} S8,
  R1 <= S6,
  P_{0,2} JOIN_{0,1:1,2} (S6 S6) <= S6,
  DIFFERENCE (S2 S1) <= T1,
  CROSS (P_{0} S1 P_{1} T2) <= T1,
  UNION (T1 S3) <= T2,
  T1 <= UNION (T2 S4),
  P_{0,4} JOIN_{0,1:1,2:2,3:3,4}
                  (S5 S5 S5 S5) <= T6,
  P_{3,5} S_{0=2,1=3} CROSS(S7 S8 S8) <= T5,
  S6 <= T4
ELIMINATE
  S1,S2,S3,S4,S5,S6,S7,S8
;
ELIMINATED S1, S2, S3, S4, S5, S7
YIELDING
  P_{3,5} S_{0=2,1=3} X (R5 S8 S8) <= T5,
  P_{0,4} P_{0,1,3,5,7} S_{1=2,3=4,5=6} X (
    UNION(P_{0,1} S_{0=2,2=3} X (R2 R2)
      P_{0,1} S_{0=3,1=2} X (R2 R2))
    UNION(P_{0,1} S_{0=2,2=3} X (R2 R2)
      P_{0,1} S_{0=3,1=2} X (R2 R2))
    UNION(P_{0,1} S_{0=2,2=3} X (R2 R2)
      P_{0,1} S_{0=3,1=2} X (R2 R2))
    UNION(P_{0,1} S_{0=2,2=3} X (R2 R2)
      P_{0,1} S_{0=3,1=2} X (R2 R2))) <= T6,
  T1 <= T2,
  X (P_{0} P_{0,2}
    LEFTOUTERJOIN_{0,1:1,2}(R2 R3)
     P_{1} T2) <= T1,
  P_{0} R5 <= P_{0} S8,
  P_{1} R5 <= P_{1} S8,
  R1 <= S6,
  P_{0,2} J_{0,1:1,2}(S6 S6) <= S6,
  S6 <= T4
```

The output is structured into two parts: the statement of the composition problem (before the semicolon) followed by the result of composition. The problem statement comprises three sections: SCHEMA, CONSTRAINTS, and ELIMINATE. The SCHEMA section lists the relation symbols used in the schemas σ_1, σ_2, and σ_3, and their arities. For example, "R1(2)" states that "R1" is a binary relation symbol.

[3] http://www.research.microsoft.com/db/ModelMgt/composition/.

The CONSTRAINTS section holds the union of mapping constraints from m_{12} and m_{23}. The symbols from σ_2 are listed in the ELIMINATE section. The result of composition appears in sections ELIMINATED and YIELDING. The ELIMINATED section lists the symbols that were eliminated successfully, while the YIELDING section contains the output constraints for the composition mapping m_{13}.

The EBNF syntax of mapping constraints is shown below, for the relational operators used in the sample run. The operator names "PROJECT", "SELECT", and "CROSS" can be abbreviated as "P", "S", and "X" (the respective EBNF productions are omitted for brevity):

```
constraint := expr "<=" expr
            | expr "=" expr

expr := relation
    | "PROJECT_{" slots "}" expr
    | "SELECT_{" cond ("," cond)* "}" expr
    | "CROSS" exprList
    | "JOIN_{" slots (":" slots)* "}" exprList
    | "LEFTOUTERJOIN_{" slots ":" slots "}" exprPair
    | "DIFFERENCE" exprPair
    | "UNION" exprList

exprPair := "(" expr expr ")"
exprList := "(" expr+ ")"
slots    := slot ( "," slot )*
cond     := slot "=" (slot | const)
```

To illustrate the slotted notation, suppose that the binary relations R_2 and R_3 have the signatures R_2(A, B) and R_3(C, D). Then, the first input constraint of the sample run states:

$$S_1 = \pi_{A,D}(R_2 ⟕_{B=C} R_3)$$

We used two data sets in our experiments. The first one contains 22 composition problems drawn from the recent literature [4,6,7], which illustrate subtle composition issues. The attractiveness of this data set is that the expected output mappings were verified manually and documented in the literature, sometimes using formal proofs. So, this data set serves as a test suite that can be used for verifying implementations of composition.

We found that the output constraints produced by our algorithm are often more verbose than the ones derived manually, i.e., simplification of output mappings is essential. An example of such simplification is detecting and removing implied constraints. Mapping simplification appears to be a problem of independent interest and is out of scope of this paper. Furthermore, we found that our technique of representing key constraints using the active domain symbol works well in many cases, but fails in others due to de-Skolemization. Leveraging key constraints in a more direct way, e.g., using specialized rules in the composition algorithm, may increase its coverage.

The second data set used in the experiments is synthetic. Using synthetic mappings appears to be the primary way to study the scalability and effectiveness of mapping composition; this approach was also followed in [13]. Our study is based on the schema evolution scenarios outlined in the introduction. Specifically, we focus on schema editing and schema reconciliation tasks since mapping adaptation can be viewed as a special case of schema reconciliation where one of the input mappings is fixed.

In the study based on synthetic mappings, we determine the success rate of our algorithm in eliminating symbols for various composition tasks, measure its execution time, and investigate the contributions of the main steps of the algorithm on its overall performance. To generate synthetic data for our experiments, we developed a tool which we call a *schema evolution simulator*. The simulator is driven by a weighted set of schema evolution primitives, such as adding or dropping attributes and relations, schema normalization, or vertical partitioning, and produces a mapping between the original schema and the evolved schema.

In the schema editing scenario, we run the simulator to mimic the schema transformation operations performed by a database designer. The mapping between the original schema and the current state of the schema is composed with the mapping produced by each subsequent schema evolution primitive. We record the success or failure of each composition operation for the applied primitives.

To study schema reconciliation, we use the simulator to produce a large number of evolved schemas and mappings for a given original schema. We then compose the generated mappings pairwise using our composition tool.

Our key observations from this study are summarized below:

- Our algorithm is able to eliminate 50–100% of the symbols in the examined composition tasks (Figs. 1, 4–7).
- The algorithm's median running time is in the subsecond range for mappings containing hundreds of constraints (Figs. 2–4, 6, 7).
- Composition becomes increasingly harder as more schema evolution primitives are applied to schemas (Fig. 6).
- Certain kinds of schema evolution primitives are more likely to produce complications for composition than others (Figs. 1, 2, 4).
- Key constraints do not substantially affect the symbol-eliminating power of the algorithm, yet significantly increase the running time (Figs. 1, 2, 7).
- It is beneficial to keep the symbols that could not be eliminated in the mappings as second-order constraints as long as possible. Subsequent composition operations may eliminate up to a third of those symbols (Fig. 7).
- Our algorithm appears to be order-invariant on the studied data sets, i.e., it eliminates the same fraction of symbols no matter in what order the symbols are

Table 2 Schema evolution primitives

Primitive	Description	Input relation	Output relation(s)	Mapping constraint(s)
AR	Add relation	$\emptyset$	$R(\underline{\mathbf{A}}, \mathbf{B})$	$\emptyset$
DR	Drop relation	$R(\mathbf{A})$	$\emptyset$	$\emptyset$
AA	Add attribute	$R(\mathbf{A})$	$S(\mathbf{A}, C)$	$R = \pi_{\mathbf{A}}(S)$
DA	Drop attribute	$R(\mathbf{A}), C \in \mathbf{A}$	$S(\mathbf{A} - \{C\})$	$\pi_{\mathbf{A}-\{C\}}(R) = S$
D	Add default	$R(\mathbf{A})$	$S(\mathbf{A}, C)$	$R \times \{c\} = S$; $R = \pi_{\mathbf{A}}(\sigma_{C=c}(S))$
H	Horizontal partitioning	$R(\mathbf{A}), C \in \mathbf{A}$	$S(\mathbf{A}), T(\mathbf{A})$	$\sigma_{C=c_S}(R) = S$; $\sigma_{C=c_T}(R) = T$; $R = S \cup T$
V	Vertical partitioning	$R(\underline{\mathbf{A}}, \mathbf{B}, \mathbf{C})$	$S(\underline{\mathbf{A}}, \mathbf{B}), T(\underline{\mathbf{A}}, \mathbf{C})$	$\pi_{\mathbf{A},\mathbf{B}}(R) = S$; $\pi_{\mathbf{A},\mathbf{C}}(R) = T$; $R = S \bowtie_{\mathbf{A}} T$
N	Normalization	$R(\mathbf{A}, \mathbf{B}, \mathbf{C})$	$S(\underline{\mathbf{A}}, \mathbf{B}), T(\mathbf{A}, \mathbf{C})$	Same as vertical; $\pi_{\mathbf{A}}(T) \subseteq \pi_{\mathbf{A}}(S)$
Sub	Subset	$R(\mathbf{A})$	$S(\mathbf{A})$	$R \subseteq S$
Sup	Superset	$R(\mathbf{A})$	$S(\mathbf{A})$	$R \supseteq S$

tried (in the loop in Line 3 of procedure COMPOSE in Sect. 3.1).

In the remaining space we present some details of our study. The experiments were conducted on a 1.5 GHz Pentium M machine.

4.2 Schema evolution simulator

Table 2 lists the schema evolution primitives implemented in our simulator. Each primitive takes zero or one relation as input, and produces zero or more new relations and constraints. The produced constraints link the output relations to the input relations, or represent key or inclusion constraints on the output relations. In the figure, we use the named perspective on the relational algebra to simplify the exposition. Attribute lists are shown in bold (e.g., $\mathbf{A}$), keys are underlined, and lower case symbols denote constants. The shown list of primitives covers a large fraction of those used in schema evolution and data integration literature but we do not claim completeness.

The first four primitives AR, DR, AA, and DA add or delete relations and attributes. Primitive AR creates a new relation. The arity of the new relation is chosen at random between some minimal and maximal relation arity (2 and 10, in our study). If keys are enabled, the created relation may have a key whose size is chosen between some minimal and maximal value (1 and 3, in our study). Primitive AA adds a new attribute C to the relation R, i.e., produces a new relation S that contains C and all existing attributes $\mathbf{A}$ of R. The mapping constraint $R = \pi_{\mathbf{A}}(S)$ states that R can be reconstructed as a projection on S. Primitive DA is complementary to AA.

Primitives D, H, V, and N have forward and backward variants (an explicit list of all supported primitives appears in Table 3). The forward variant, labeled with subscript 'f', contains only the constraints that define the output relations

Table 3 Event vectors

Primitive	Default	Fwd	Bwd	Preserving
AR	1	1	1	1
DR	0.2	0.2	0.2	0
AA	2	2	2	2
DA	1	1	1	0
$\mathsf{D_f}$, $\mathsf{H_f}$, $\mathsf{V_f}$, $\mathsf{N_f}$, Sub	1	1	0	0
$\mathsf{D_b}$, $\mathsf{H_b}$, $\mathsf{V_b}$, $\mathsf{N_b}$, Sup	1	0	1	0
D, H, V, N	1	0	0	1

in terms of the input relations. The backward variant ('b') contains only the constraints that define the inputs in terms of the outputs. The forward and backward variants reflect distinct evolution scenarios, as we explain below in more detail.

Primitive D adds an attribute C with a default value c. The forward variant $\mathsf{D_f}$ outputs the mapping constraint $R \times \{c\} = S$, while $\mathsf{D_b}$ outputs $R = \pi_{\mathbf{A}}(\sigma_{C=c}(S))$. Thus, $\mathsf{D_f}$ states that S is determined by R and that the newly added attribute C contains c-values only. $\mathsf{D_b}$ produces a weaker constraint that allows attribute $S.C$ to contain other values beyond c.

Primitive H performs a lossless horizontal partitioning of the input relation R. The forward variant $\mathsf{H_f}$ outputs the mapping constraints $\sigma_{B=c_S}(R) = S$, $\sigma_{B=c_T}(R) = T$, which tell us how to obtain partitions S and T by selecting tuples from R. Some data from R is allowed to be lost. The backward variant $\mathsf{H_b}$ produces constaint $R = S \cup T$, which states that R can be reconstructed from S and T but does not include the partitioning criterion. The constraints produced by $\mathsf{H_f}$ and $\mathsf{H_b}$ do not imply each other.

The vertical partitioning primitives V, $\mathsf{V_f}$, $\mathsf{V_b}$ require the input relation R to have a key. The attribute set of R is partitioned across output relations S and T.

Primitive N captures the schema normalization rule from database textbooks. The input relation R, which may contain

redundant data, is split into S and T. **A** becomes a key in S, and a foreign key in T. This primitive assumes the existence of an implicit functional dependency $\mathbf{A} \rightarrow \mathbf{B}$ on R. Although such functional dependencies are not enforced by commercial database systems and hence are missing in real-world schemas, database administrators use this implicit knowledge in schema design. Should this implicit dependency be violated for a particular instance of R, then applying the $\mathsf{N_f}$ primitive incurs data loss, while N produces an inconsistency.

The constants c, c_T, and c_S used in the primitives D and H and their variants are drawn from a fixed-size pool of constants (in our study, of size 10).

All evolution primitives discussed above produce equality constraints. Some schema integration and data exchange scenarios in the literature assume the so-called open-world semantics for mapping constraints: informally speaking, subset is used in place of equality. To accommodate these scenarios in our experiments, we added two further primitives, Sub and Sup. Composition of mapping constraints produced by Sub and Sup with those of other primitives yields inclusion constraints that generalize global-local-as-view (GLAV) settings. For example, applying the DA primitive produces the constraint $\pi_{\mathbf{A}-\{C\}}(R) = S$. Applying the Sub primitive thereafter may produce the constraint $S \subseteq T$. Composing the two yields the constraint $\pi_{\mathbf{A}-\{C\}}(R) \subseteq T$.

Event vectors In each run, the schema evolution simulator applies a sequence of *edits*. Each edit consists of executing a particular schema evolution primitive followed by mapping composition.

An *event vector* specifies the proportions of primitives of a certain kind appearing in an edit sequence. In our study we use four event vectors that are depicted in Table 3. For example, in the Default vector the proportion of AR is 1, the proportion of AA is 2 meaning that an attribute is added twice as often as a relation. That is, in an edit sequence of size 100 the AR primitive is applied on average $1 \cdot 100 \div (0.2+2+1 \cdot 16) \approx 5.5$ times.

We are not aware of studies that investigated the frequency of schema evolution primitives used in real-world evolution scenarios. Thus, we assume that all primitives are applied with the same frequency, with the exception of adding attributes (AA is twice as frequent) and dropping relations (DR is five times less frequent).

The Default event vector excercises all schema evolution primitives. In contrast, the Forward vector focuses on those primitives where the input relations determine the output relations. Intuitively, the mapping produced by using the Forward vector defines the evolved schema as a view on the original schema. In the Backward vector, the original schema is defined as a view on the evolved schema. The Preserving vector contains only the 'lossless' schema evolution primitives; all information of the original schema is preserved in the evolved schema and vice-versa.

4.3 Study on synthetic mappings

The output mapping produced by the composition algorithm may be exponentially larger than the input mappings. To control the exponential blowup, the algorithm aborts whenever the output-to-input size ratio exceeds a certain factor (100, in our study). The size of mappings is measured as the total number of operators across all constraints. In our experiments, the algorithm fails to eliminate only about 1% of symbols due to such blowup.

Schema editing scenarios Figure 1 shows the success rate of our algorithm in composing the mappings of an edit sequence. Four configurations are examined ('no keys', 'keys', 'no unfolding', 'no right compose'). The data for each configuration was obtained as follows: in each run, 100 edits are applied to a randomly generated schema of a default size 30. The mappings produced in each edit are composed. The proportions of primitives in the edit sequence correspond to the Default event vector. The numbers shown for each configuration are averaged across 100 runs. That is, 10,000

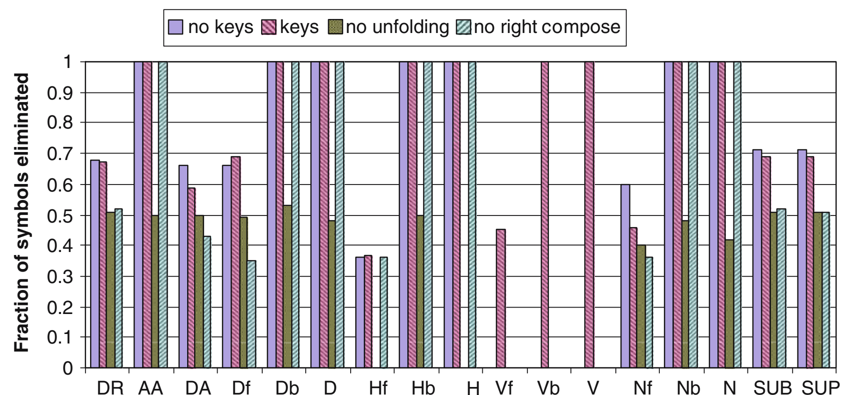


Fig. 1 Eliminated symbols per primitive

Table 4 Parameters of experimenal setup

Parameter	Value
Minimal initial relation arity	2
Maximal initial relation arity	10
Minimal initial key size	1
Maximal initial key size	3
Size of constant pool used in primitives	10
Exponential blowup threshold factor	100
Default schema size	30
Default size of edit sequence	100
Number of edit runs to average over	100
Number of composition runs to average over	500

composition tasks are executed for each configuration. Table 4 summarizes the key parameters used in our study.

In the ‘no keys’ and ‘keys’ configurations all features of the algorithm are exploited; in the latter the relations may contain keys. In the ‘no unfolding’ configuration, the View Unfolding module of the algorithm is disabled. Similarly, for ‘no right compose’. Vertical partitioning is not applicable if no keys are present, therefore the respective bars remain empty in all configurations but ‘keys’.

The figure shows that the symbols introduced by some primitives are easier to eliminate than others. H_f, V_f, and N_f are the ‘hardest’ primitives. Adding keys to schemas does not significantly affect the symbol-eliminating power of the algorithm. Turning off view unfolding or right compose weakens the algorithm substantially.

The execution times for this experiment are in Fig. 2. Disabling view unfolding or adding keys increases the running time significantly. When a keyed relation is eliminated, its key constraints need to be propagated to all expressions that reference it. So, mappings produced in the ‘keys’ setting contain on average 218 constraints with 4,300 relational operators, as opposed to 95 constraints with 800 operators for ‘no keys’. The median execution time per run (for all 100 edits) are around 0.2 s for ‘no keys’ and ‘no right compose’, 2.8 s for ‘keys’, 2.1 sec for ‘no unfolding’ — a tenfold increase (not shown in the figure). The reason for reporting the median time instead of the average time is exemplified in Fig. 3: most runs have close running times except for a few outliers that skew the average. This graph is characteristic across all experiments.

Figure 4 illustrates the impact of inclusion constraints (vs. equality constraints) on a few selected schema evolution primitives and the overall running time. Each edit sequence corresponding to value x on the x-axis is constructed by using the event vector obtained as a copy of the Default vector in which the proportion of Sub and Sup primitives is set to x. On average, the composition tasks become more difficult (the total fraction of symbols eliminated in all edits decreases) since the effectiveness of view unfolding drops. However, the algorithm fails faster as it detects the symbols that cannot be isolated on either the left or right side of constraints, and so the overall running time decreases.

Schema reconciliation scenarios Figure 5 depicts a schema reconciliation scenario. Each task consists of composing two mappings produced by the simulator for two sequences of 100 edits each. To obtain first-order input mappings, only those edit sequences produced by the simulator were considered in which all symbols were eliminated successfully.

The data points shown in the figure were obtained by averaging over 500 composition tasks. Increasing the size of the intermediate schema (which contains the symbols to be eliminated) simplifies the composition problem. This is an expected result since the simulator applies the edits to randomly chosen relations, so a larger intermediate schema makes it less likely that the constraints in the two input mappings interact, i.e., mention the same symbols. For this same reason, increasing the number of edits makes the composition problem harder; the fraction of eliminated symbols drops while the running time increases (see Fig. 6).

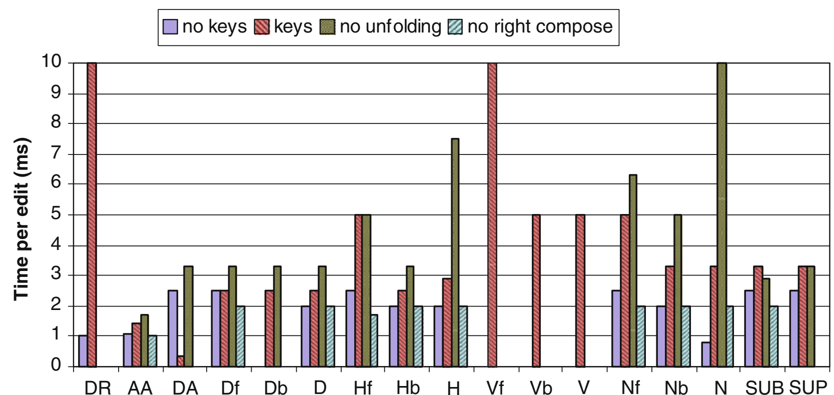


Fig. 2 Execution time for each primitive

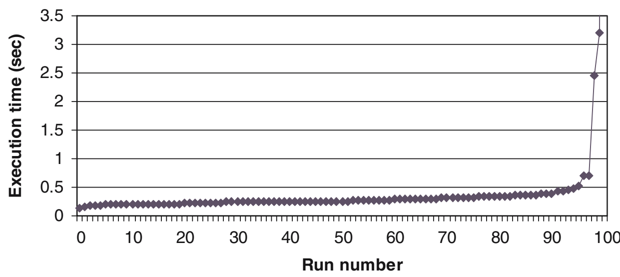


Fig. 3 Sorted execution time across 100 runs for ‘no keys’

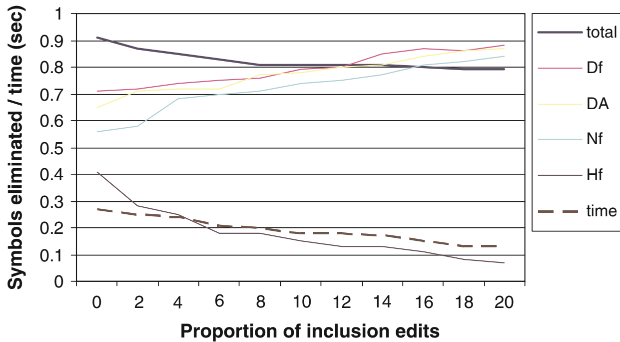


Fig. 4 Increasing proportion of inclusion primitives

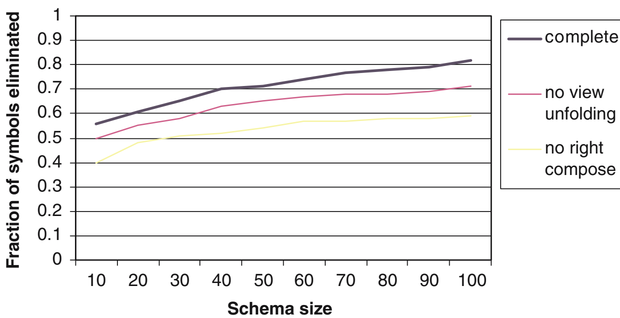


Fig. 5 Varying schema size

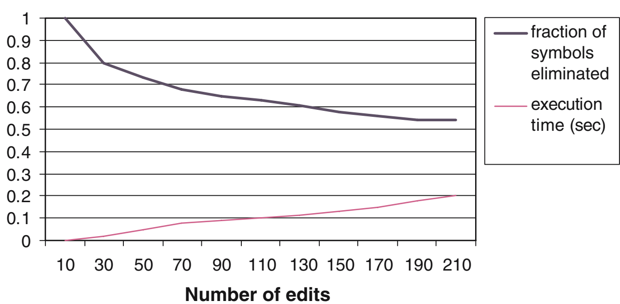


Fig. 6 Varying number of edits

Disabling view unfolding or right compose has a similar effect on the composition performance as in the schema editing scenarios: as shown in Fig. 5, 10–20% fewer symbols get eliminated. In addition, the execution time for disabling view unfolding increases by about an order of magnitude (not shown in the figure). Disabling left compose (not shown) does not have a noticeable impact of the symbol-eliminating power of the algorithm for the settings used in the study, mostly because the simulator does not introduce any relational operators beyond $\sigma, \pi, \cup, \bowtie, \times$.

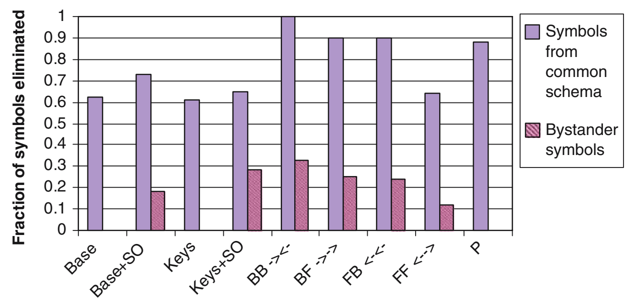


Fig. 7 Impact of second-order, keys, and event vectors

Mapping composition may necessarily fail to eliminate symbols from the intermediate schema [4,7]. As illustrated in Fig. 7, it is be beneficial to keep those symbols around as ‘bystanders’ in second-order constraints even if ultimately we are interested in obtaining first-order mappings. We found that subsequent mapping compositions (or other operations on mappings) may facilitate symbol elimination at a later stage. To quantify this effect, we configured the simulator to retain second-order mappings, which may contain symbols that could not be eliminated upon edits.

Figure 7 illustrates a schema reconciliation problem for schema size 30 and input mappings generated for edit sequences of length 100. No keys are used in the Base configuration and the input mappings are first-order. In Base+SO, second-order mappings are allowed. The figure shows that close to 20% of second-order symbols in the input mappings can be eliminated in the final composition. Moreover, going second-order increases the number of symbols from the intermediate schema that can be eliminated successfully. A similar effect is observed when keys are allowed.

Finally we discuss the impact of varying the event vectors. The labels B, F, and P in Fig. 7 correspond to the event vectors Bwd, Fwd, and preserving of Table 3. For example, in task BB the Bwd vector is used for generating both input mappings. Consequently, BB corresponds to composing mappings each of which is a view pointing inward, i.e., expressing the intermediate schema in terms of an evolved schema. Close to 100% of symbols from the intermediate schema get eliminated in this setting, and about a third of bystander symbols for second-order mappings. In BF and FB tasks, the mappings are views pointing in the same direction. If the views point to the opposite directions (FF), composition turns out to be harder. In contrast, close to 90% of symbols get eliminated for the information-preserving vector P;

the bystander bar is not shown since the input mappings produced by the simulator do not contain any second-order constraints.

5 Conclusions

This paper presented a new algorithm for composition of relational algebraic mappings that extends significantly the algorithms in [4,7]. It has many new features: it makes a best effort to eliminate as many symbols as possible; it can handle unknown or partially known operators, including ones that are not monotonic in all of their arguments; it introduces the left-compose transformation; and it is highly extensible. We demonstrated its value by an experimental study of its effectiveness on a large set of synthetically-generated mappings.

References

1. Bernstein, P.A.: Applying model management to classical meta-data problems. In: CIDR (2003)
2. Bernstein, P.A., Halevy, A.Y., Pottinger, R.: A vision of management of complex models. SIGMOD Record **29**(4), 55–63 (2006)
3. Buneman, P., Davidson, S.B., Kosky, A.: Theoretical aspects of schema merging. In: Proc. EDBT, pp. 152–167 (1992)
4. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.C.: Composing schema mappings: second-order dependencies to the rescue. ACM TODS **30**(4), 994–1055 (2005)
5. Madhavan, J., Halevy, A.Y.: Composing mappings among data sources In: Proc. VLDB, pp. 572–583 (2003)
6. Melnik, S., Bernstein, P.A., Halevy, A., Rahm, E.: Supporting Executable Mappings in model management. In: Proc. ACM SIGMOD (2005)
7. Nash, A., Bernstein, P.A., Melnik, S.: Composition of mappings given by embedded dependencies. ACM TODS **32**(1) (2007)
8. Pottinger, R., Bernstein, P.A.: Merging models based on given correspondences. In: Proc. VLDB, pp. 826–873 (2003)
9. Stonebraker, M.: Implementation of integrity constraints and views by query Modification. In: Proc. ACM SIGMOD, pp. 65–78 (1975)
10. Tatarinov, I., Halevy, A.Y.: Efficient query reformulation in peer-data management systems. In: Proc. ACM SIGMOD (2004)
11. Taylor, N., Ives, Z.: Reconciling changes while tolerating disagreement in collaborative data sharing. In: Proc. ACM SIGMOD (2006)
12. Yannakakis, M.,: Papadimitriou, C.H.: Algebraic dependencies. J. Comput. System. Sci. **25**(1), 2–41 (1982)
13. Yu, C., Popa, L.: Semantic adaptation of schema mappings when schemas evolve. In: Proc. VLDB, pp. 1006–1017 (2005)