The VLDB Journal
DOI 10.1007/s00778-009-0159-9

REGULAR PAPER

Schema mapping and query translation in heterogeneous P2P XML databases

Angela Bonifati · Elaine Chang · Terence Ho · Laks V. S. Lakshmanan · Rachel Pottinger · Yongik Chung

Received: 18 September 2008 / Revised: 1 May 2009 / Accepted: 9 July 2009

Abstract Peers in a peer-to-peer data management system often have heterogeneous schemas and no mediated global schema. To translate queries across peers, we assume each peer provides correspondences between its schema and a small number of other peer schemas. We focus on query reformulation in the presence of heterogeneous XML schemas, including data–metadata conflicts. We develop an algorithm for inferring precise mapping rules from informal schema correspondences. We define the semantics of query answering in this setting and develop query translation algorithm. Our translation handles an expressive fragment of XQuery and works both along and against the direction of mapping rules. We describe the HePToX heterogeneous P2P XML data management system which incorporates our results. We report the results of extensive experiments on HePToX on both synthetic and real datasets. We demonstrate our system utility and scalability on different P2P distributions.

Keywords Schema mapping · XML query translation · Heterogeneous Peer-to-Peer XML databases

A. Bonifati (✉)
Icar-CNR, Via P. Bucci 41/C, 87036 Rende (CS), Italy
e-mail: bonifati@icar.cnr.it

E. Chang · T. Ho · L. V. S. Lakshmanan · R. Pottinger · Y. Chung
UBC, 2366 Main Mall, Vancouver, BC V6T 1Z4, Canada
e-mail: echang@cs.ubc.ca

T. Ho
e-mail: terenho@cs.ubc.ca

L. V. S. Lakshmanan
e-mail: laks@cs.ubc.ca

R. Pottinger
e-mail: rap@cs.ubc.ca

Y. Chung
e-mail: ychung25@cs.ubc.ca

1 Introduction

A peer-to-peer data management system (PDMS) (e.g., [5,8,12,20,28,37,44]) is an ad-hoc collection of independent peers that have formed a network in order to map and share their data. For example, consider a group of hospitals that need to translate patient data as patients move back and forth between hospitals; Figure 1 shows the schemas of two such hospitals `Montreal General Hospital` (`MON` for short) and `Boston General Hospital` (`BOS` for short).[1]

As the network is ad-hoc and lacks an overriding authority, peers in a PDMS typically have heterogeneous schemas, and there is no mediated, global schema. Therefore, there must be a way for data to be translated between the peers' schemas. Hence, PDMSs assume that each peer provides correspondences between its schema and a small number of other peer schemas, known as *acquaintances*. Creating such a mapping is a difficult task, as the schemas may differ substantially, even in similar domains. An example of such differences is shown in Fig. 1, where two heterogeneous schemas are depicted, the `MON` schema in Fig. 1a and the `BOS` schema in Fig. 1b. Before explaining the two schemas, we briefly introduce the notation. We use here a simple graph representation of schemas. This representation is rich enough to capture atomic types, sets, tuples, nested structures, keys, referential constraints, and optionality. In particular, we denote with a solid black edge the relationship between a parent element and a child element in the schema, the edge being labeled with a cardinality constraint ('*' and '+' to indicate multiplicity) or with an optionality constraint ('?'), or unlabeled otherwise (to indicate '1–1' cardinality). A solid black edge labeled with '@' indicates the relationship between a

[1] This figure is adapted from [8,20,33].

Published online: 21 August 2009

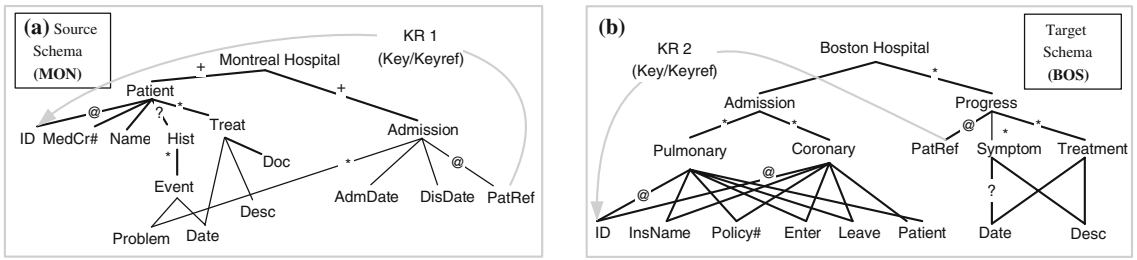


Fig. 1 Example of heterogeneous peer schemas for **a** `Montreal General Hospital` and **b** `Boston General Hospital`

parent element and a child attribute in the schema. A solid gray arrow is used to represent the referential integrity constraint on Keys and foreign Keys and labeled with '(Key/KeyRef)'. In the `MON` schema, every patient is assigned a unique `ID` by the hospital and has a unique Medicare number (`MedCr#`). Symptoms of problems experienced by patients and the treatments they are administered are all grouped under patients. Patient admission data is maintained separately and is captured via the Key/KeyRef link `@PatRef` → `@ID` (the gray arrow KR1 in Fig. 1a). On the other hand, the `BOS` database in Boston (Fig. 1b) is organized quite differently: at admission time, patients are classified on the basis of their main complaint (pulmonary and coronary, as shown in the figure). The usual patient details, such as name, ID, etc. are stored under this classification. Progress of patients during their stay in the hospital is recorded: patients' history of health problems and the treatments administered are tracked. All of this information is connected to the patients via the Key/KeyRef link `@PatRef` → `@ID` (the solid gray arrow KR2 in Fig. 1b).

When patients move between `MON` and `BOS`, their records must move with them. However, given the differences in the schemas, this is far from being trivial. In particular, translating queries over one source into queries over another requires (1) creating an unambiguous mapping that precisely reflects the transformation between the two sources and (2) building a system that automatically does the query translation. Since the source schema `MON` and the target schema `BOS` are to be used interchangeably, query translation must be possible both along and against the direction of the mappings.

Besides asking a specific peer for data, a further goal is to permit users and applications of any peer database to access data items of interest by simply posing a query to their local peer, regardless of the location of the data items or the schema under which they are organized. That is, the existence of numerous peers and their schemas should be transparent to the user/application.

To give a more concrete example, to answer the query "what are the treatments administered to patients admitted with a coronary illness?", data from all peers should be accessible to the original peer. Since these peers have different schemas, queries posed to a peer need to be translated appropriately to run on other peers; this underscores the need for support for mapping creation and query translation.

To achieve these goals, we present the HEterogeneous Peer TO peer Xml database system (HePToX[2]), a P2P Heterogeneous XML database system. Previous research on P2P query reformulation systems has considered mappings directly expressed in a run-time language, such as XQuery in the Piazza system [21]. As opposed to such proposals, HePToX is based on a simple visual user interface [11], that requires that the peer administrator provides correspondences between the local schema and the acquaintance schema using an intuitive notation of arrows and boxes, such as those shown in Fig. 2. From these simple correspondences, HePToX will automatically derive precise *mapping expressions* between the schemas, which are stated as Datalog-like rules. Using these mapping expressions, HePToX allows a peer user to query any peer's data using its own local schema and translates queries in a way that is consistent with the mapping semantics. Thus, joining a PDMS becomes a lightweight operation, though a DBA may further examine the mapping expressions and make adjustments if desired.

The visual correspondences in HePToX are arrows and boxes, illustrated in Fig. 2, and explained in Sect. 2. Such correspondences may be manually provided by each peer administrator, or simply output by (semi-) automatic schema matching tools [16]. Crucially, the input correspondences to HePToX includes a construct (namely, the box) that allows the inference of data–metadata correspondences. Such correspondences were initially exploited in our demo proposal [11], and have recently been studied as a useful extension of mappings in Clio [24]. To the best of our knowledge, our work remains the first effort toward the use of such correspondences in query reformulation in a P2P scenario.

Although Sect. 7 provides a full comparison of related work, we briefly highlight some key differences between HePToX and previous work here. Differences in data representation in P2P networks were addressed in [28] by introducing mapping tables: e.g., we might specify that ID GDB:123 in one database corresponds to all ID-values

[2] Pronounced Hep Talk: heterogeneous peers talk!

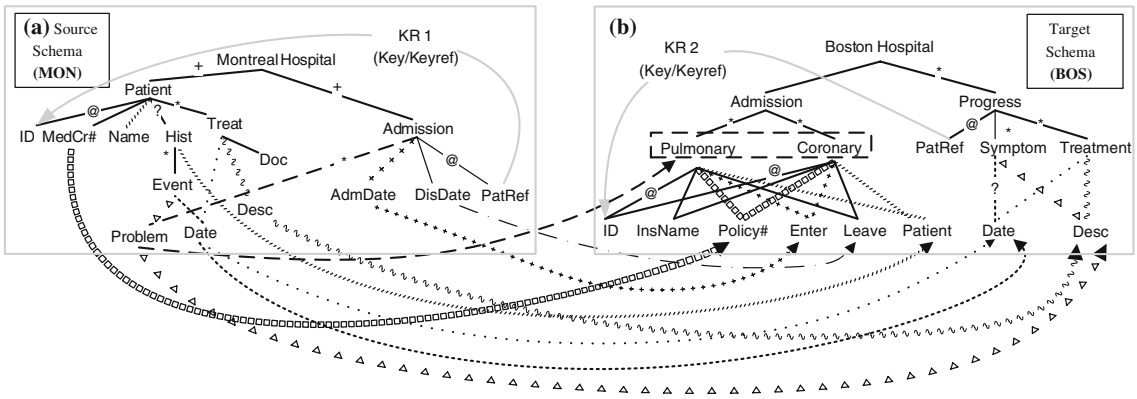


Fig. 2 Mapping schemas MON and BOS. To minimize clutter, some arrows between corresponding tags have been omitted (e.g., @ID to @ID)

but SP:456 in another database. These are aimed at mapping value aliases, which are orthogonal to schema mappings.

The lines and the mappings specified in HePToX are similar to those used in Clio [32,37]. Clio is a data exchange tool, and focuses on how to efficiently compute a target instance provided a source instance, source and target schemas, and a set of mappings between them. In contrast, HePToX is a *query reformulation tool*, expressly designed for PDMS applications. In such setting, a target instance already exists, so HePToX focuses on how to translate queries across the peers heterogeneous schemas. While we present a thorough comparison between the two systems in Sect. 7, it is worth noting the following: in principle, one could translate the data instance from every peer to the querying peer and then answer the query there. But this is not practical, as it would require expensive translation of every peer's data to every other peer's schema.

Query rewriting over the mappings is studied in [46], where source-to-target mappings are considered and queries must be translated from the target schema to the source schema. In contrast, HePToX focuses on translation of queries from the source schema to the target schema, which is not handled in [46] and is a more difficult problem. While we also translate queries from the target schema to the source schema, we handle a less general case, with flat queries and no constraints on the target. Moreover, we can seamlessly cover data–metadata mappings in both directions of the translation, which was not done in [46]. We refer the reader to Sect. 7 for a detailed discussion.

We develop a forward query translation algorithm that is based on query answering using views [29]. To the best of our knowledge, this is the first query translation algorithm that deals with schema mappings, including data–metadata mappings along and against the direction of mappings. We also address the consequent scalability problem, which arises in a distributed setting, by deploying our translation algorithms in a P2P infrastructure. However, neither Piazza [20] nor Clio [24,46] addresses query translation across schemas involving data-metadata interplay as illustrated in Fig. 2.

In summary, we make the following contributions:

- We propose an informal mechanism for specifying correspondences using arrows and boxes.
- We develop TreeLog, a Datalog-like language used to express the rules between source and target (Sect. 3). The language queries and restructures semistructured data and elegantly handles the data-metadata interplay between schemas and schema mappings.
- We develop an algorithm for inferring mapping rules between the schemas, and discuss the class of transformations captured by the rules (Sect. 4).
- We define the semantics of peer queries. We develop a novel query translation algorithm that handles a simple but significant fragment of XQuery and show that it is correct w.r.t. the above semantics. We illustrate our algorithm with examples (Sect. 5). Translation is non-trivial even for the XQuery fragment considered and works both along and against the direction of mapping rules. Translation along the mappings is entirely new for the class of transformations addressed.
- We developed the HePToX system which incorporates the ideas in this paper. We report on extensive experiments to measure the effectiveness of our query translation algorithm, as well as the scalability of our approach, with both uniform and non-uniform schema distributions. We discuss the results and the lessons learned (Sect. 6).

Additionally, Sect. 2 gives a motivating example. Related work is discussed in Sect. 7, while Sect. 8 summarizes the paper and discusses future research.

2 A motivating example—the correspondences

This section illustrates the subtleties of the problems which HePToX considers by revisiting the example schemas in

Fig. 1 and the mapping between them in Fig. 2. In Fig. 2, each arrow between the schemas specify a correspondence between a pair of elements from the two schemas. For example, one arrow shows that `DisDate` in `MON` corresponds to `Leave` in `BOS`. To make it easier to see the correspondences, we use a different type of line for each correspondence.

HePToX considers both the internal structure and the leaf elements. This means that because the schemas have shared elements, i.e., they are DAG structured, we need to specify correspondences between parental paths if there is more than one option. For example, the element `Desc` has two parents, `Treatment` and `Symptom`, in `BOS`. To relate `Desc` in `MON` to `Treatment/Desc` in `BOS`, we use the same line type for the edges `Treat—Desc` in `MON`, `Treatment—Desc` in `BOS`, and the correspondence connecting them. Where there is no ambiguity, the correspondences propagate from the leaves up to the their parent/ancestor elements as in [16].

HePToX also allows the expression of schema to data correspondences, which is novel w.r.t. previous work on schema matching. A dashed box surrounds the schema-level concepts that are related to data level concepts in the other schema. For example, `Admission/Problem` in `MON` may contain data items, such as 'Pulmonary' or 'Coronary', so the `Pulmonary` and `Coronary` elements in `BOS` are grouped with a (thick dashed) box and are linked by an arrow to `Admission/Problem` in `MON`.

HePToX is geared to share data across different sources, thus the boxes and arrows may denote correspondences between elements of the *schemas* themselves, or between elements of the *schema* and elements of the *data instances*. The latter correspondences embody data–metadata conflicts. Notice that there is no assumption that the set of illnesses occurring in the two databases are the same or even overlap. This implies that the actual knowledge of the instances is not mandatory to the creation of such correspondences.

Arrows specify a correspondence between the identified concepts. However, arrows and boxes by themselves do not tell how data conforming to a schema may be transformed to one that conforms to the other schema. For example, in the `MON` schema, patients' history (problems and dates) and the treatments they undergo are both nested under patients. All admission information is maintained separately and linked to the appropriate patient via the patient's ID. In the `BOS` database, treatment and history information (symptoms) is separated out from patients and linked to them via their ID. Additionally, patients are represented along with the rest of the admission data, but this data is classified based on the type of problem/illness identified at the time of admission. Thus, the simple input mapping must be translated into a representation that is rich enough to adequately model these differences. In Sect. 3 we describe the mapping language that we use, and why it is expressive enough to transform instances of one of the schema into instances of the other. This transformation is closely tied to the semantics of query answering, as we will see in Sect. 5.1.

Finally, the reader may have noted that Figs. 1 and 2 only show 1–1 and data–metadata correspondences. We restricted to such kinds of correspondences in this example in order to keep the exposition simple. HePToX can handle 1–1, 1–m, and m–n correspondences, as we shall see in the remainder of the paper. As an example of 1–m mappings, the treatment of *unions of data* is a 1–m mapping, and is considered in more detail in Sect. 4. M–n mappings are a generalization of such unions.

3 The mapping language

Creating the mapping language in HePToX requires facing several challenges. In particular, it must be clean and unambiguous. Translate instances of one schema into instances of another without requiring any additional user translation than what can be inferred from the correspondences is shown in Sect. 2. However, since the goal of HePToX is to handle as many conversions as possible, the mapping language must be rich enough to handle:

- Data to metadata conversions and vice versa
- Mapping non-leaf nodes
- 1–1, 1–m, and m–n correspondences (including the mapping of unions as discussed in Sect. 4).

To create a mapping language that was able to meet all of these challenges, we adapted SchemaLog [4] to deal with tree-structured data. SchemaLog is a syntactically higher-order and semantically first-order language for querying and restructuring inter-operable relational databases. Unlike some other logic formalisms, SchemaLog could treat data and schema at par. Therefore, we extended it to express schema mappings for semi-structured data. In particular, such an extension lends itself very naturally to represent data-metadata conflicts, as we explain below.

Given the correspondences in Fig. 2, HePToX's algorithm for creating the mapping rules as in Sect. 4, will produce the mapping expressions as shown in Fig. 3. We stress that the rules and the mappings are *not* intended for *physically* transforming data from one source's schema to another. Rather, as pointed out in [5], they are intended for expressing the semantics of data exchange—if data were to be exchanged from source 1 to 2, how would it correspond to the schema of source 2. As shown in Fig. 3, these mapping rules are expressed as Datalog-like rules, (⟨rule head⟩ ⟵ ⟨rule body⟩), adapted for tree structured data. The tree expressions represent indeed the counterpart of Datalog predicates.

We first give an informal glimpse of the mapping language before providing a formal syntax and semantics. Mapping

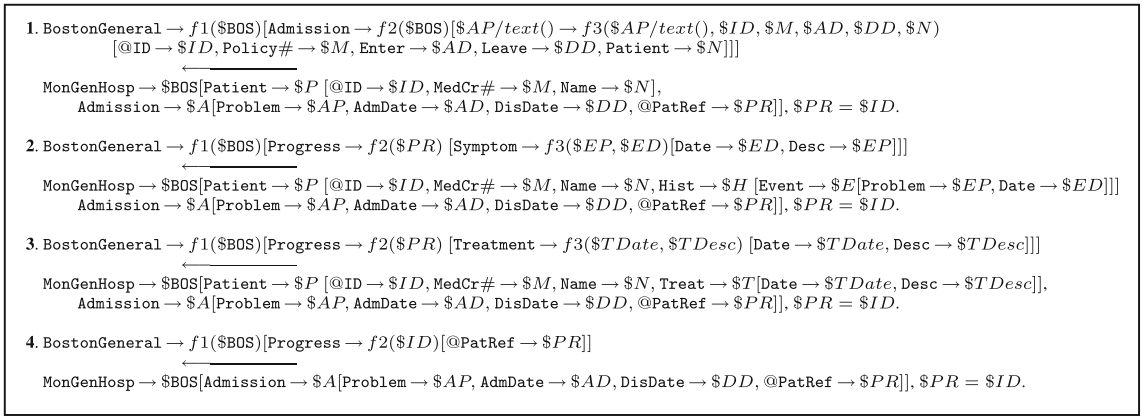


Fig. 3 Mapping rules from MON to BOS schema, shown in Fig. 2

rules are made up of *atoms* of the form $\texttt{Tag} \rightarrow id$, where Tag is a tag or a tag variable and id is the id associated with a node with this tag. Here, id may be a variable or any term of the form $f(\$v_1, \ldots, \$v_n)$, for some variables $\$v_i$ and some Skolem function f. An example appears in the next section. Similar to Clio [37], SchemaLog [4], and other previous works (e.g., [25,35]), we use Skolem functions both for creating new node ids and for grouping. Atoms can nest inside other atoms, thus expressing XML nesting. A comma-separated list of atoms expresses the sub-elements of a given element. Attributes are preceded with a '@'.

Atoms can be further nested to form *tree expressions*. Tree expressions are either atoms ($t \rightarrow i$) or are of the form $t \rightarrow i[TE_1, \ldots, TE_k]$, where $t \rightarrow i$ is an atom and TE_i are tree expressions. In Fig. 3, each rule head is a tree expression while the rule body is a conjunction of tree expressions and built-in predicates ($=, >$, etc.). We illustrate the extended version of SchemaLog that we propose by continuing with our hospital example.

The atom $\texttt{BostonGeneral} \rightarrow f1(\$\texttt{BOS})$ in rule 1 says translating the unique root $BOS of MON onto the BOS schema yields a unique root $f1(\$\texttt{BOS})$ of BOS. Similarly, there is a unique Admission node in BOS. The rule body binds the variable $\$AP$ to a patient's problem at admission time. $\$AP/text()$ extracts the text value associated with node $\$AP$. This value is used to form the tag of a new node, the id of which is $f3(\$AP/text(), \$ID, \$M, \$AD, \$DD, \$N)$, i.e., it is a function of the patient's admission time problem ($\$AP/text()$), patient id ($\$ID$), insurance policy (or Medicare) number ($\$M$), admission and discharge dates ($\$AD, \$DD$), and name ($\$N$). $f3$ illustrates a key point of the mapping rules: *the arguments of a Skolem function are exactly the mandatory single-valued sub-elements of the element they represent*.[3] In this case, the arguments of $f3$ are exactly the mandatory single-valued sub-elements of the Pulmonary and Coronary elements in BOS. We do not assume any knowledge of integrity constraints, but we can use primary keys and referential constraints as follows.

If we know the key of these elements (e.g., if we know that patient ID $\$ID$ uniquely determines patients), then we can make the node id of these elements a function of this key, e.g., $f'(\$ID)$, for some Skolem function f'.

Referential integrity constraints are used to determine equality predicates in the rules, e.g., $\$PR = \$ID$. The latter ones, if available, lets us to simplify the rules by using the same variable name.

Patient id, name, policy number, admission and discharge dates are all matched to their counterparts in BOS. Rule 4 maps @PatRef attribute in MON to @PatRef attribute in BOS. Note that@PR=@ID ensures the rule is safe and equates the @ID, @PatRef attributes in the BOS schema.

Rule 2 maps the patient history consisting of Problems and their Dates of occurrence (nested in MON through Hist/-Event) to Symptom/Desc and Symptom/Date in BOS. Note that in BOS, the Symptom elements are nested inside a Progress element, which has as its id a function of the patient ID (via @PatRef), i.e., $f2(\$PR)$, $\$PR = \$ID$. Thus, there is one Progress element per patient. Consequently, Symptoms are grouped by patient ID. The node ID $f3(\$EP, \$ED)$, which is used for Symptom elements, shows that for each occurrence of a problem for a given patient, a separate Symptom element is created.

Rule 3 maps treatment information from MON to BOS. Progress elements are created with id $f2(\$PR)$ just as they are in rule 2. Note that the use of the node id $f3(\$TDate, \$TDesc)$ for Treatment ensures that for every treatment on any date administered to a given patient, the corresponding Treatment element is nested inside the Progress element associated with the patient.

Node ids play a key role: e.g., Progress elements are created by rules 2 and 3 independently. Whenever the id of a Progress node created by rule 2 matches the one created

[3] i.e., They are not labeled '?', '*', or '+'.

by rule 3, they refer to one and the same node. For instance, suppose $p5$ is the ID value of a patient. Then the subtree rooted at the `Progress` node $f2(p5)$ created by rule 2 and the subtree rooted at the `Progress` node $f2(p5)$ created by rule 3 are both glued at the node $f2(p5)$. More generally, whenever subtrees are created by applications of the same or different rules, conceptually all these subtrees are glued together at roots having a common node id. This ensures that the pieces "computed" by rules are correctly glued together.

3.1 Formal semantics of mapping language

This section formally describes the syntax and semantics of *TreeLog*, the mapping language used for HePToX.

TreeLog has a vocabulary consisting of constants, variables, and function symbols of various arities. Terms are defined as in standard first-order logic. Additionally, whenever X is a variable, we define $X/\texttt{text}()$ to be a term.

An *atom* has the form $\texttt{TagTerm} \rightarrow \texttt{IdTerm}$, where: (i) `TagTerm` is either a variable, a constant, or a term of the form $X/\texttt{text}()$, where X is a variable,[4] and (ii) `IdTerm` is a standard first-order term made of constants, variables, and function symbols. A *tree expression* has the form A or $A[TE_1, \ldots, TE_k]$, where A is an atom and TE_i are tree expressions. The complete language is obtained by closing the atoms w.r.t. the connectives $\neg, \wedge, \vee$ and quantifiers $\exists, \forall$.

The semantics of this logic is based on structures. A structure is a finite node-labeled unordered forest, where leaves contain text data. More formally, a *structure* is a tuple $\mathcal{M} = (U, \lambda, R, \tau)$, where U is a non-empty domain of individuals, consisting of nodes, data values, and labels, R is a binary relation on U such that it forms an unordered forest of nodes, and τ, λ are many–one binary relations on U such that τ is defined only on the leaves of the forest and λ is defined on every node of the forest. When $R(u, v)$ holds in $\mathcal{M}$, we say v is a child of u. When $\lambda(u) = l$, we say l is the label of u. When $\tau(u) = c$, we say c is the data value at u.

Let $\mathcal{M}$ be a structure and ν be a variable instantiation that maps variables to the individuals in $\mathcal{M}$. Then we define satisfaction of formulas as follows. For any term t, we denote by $\nu(t)$ the result of uniformly replacing every variable X in t with its instantiation under ν. We write $\mathcal{M} \models F[\nu]$ to indicate the formula F is true in $\mathcal{M}$ under instantiation ν.

- For an atom $T \rightarrow I$, $\mathcal{M} \models T \rightarrow I[\nu]$ iff $\mathcal{M}$ contains a node $\nu(I)$ and a label $\nu(T)$ such that $\lambda(\nu(I)) = \nu(T)$.
- For a tree expression $T_1 \rightarrow I_1[T_2 \rightarrow I_2[TE]]$, where TE is any tree expression, $\mathcal{M} \models T_1 \rightarrow I_1[T_2 \rightarrow I_2[TE]][\nu]$ iff $\mathcal{M} \models T_1 \rightarrow I_1[\nu]$, and $\mathcal{M} \models T_2 \rightarrow I_2[\nu]$, and $\nu(I_2)$ is a child of $\nu(I_1)$ and $\mathcal{M} \models T_2 \rightarrow I_2[TE][\nu]$.
- Satisfaction w.r.t. conjunction, disjunction, negation, and implication of formulas is defined in the standard way.
- For a tree expression $A[TE_1, \ldots, TE_k]$, where A is an atom and TE_i are tree expressions, $\mathcal{M} \models A[TE_1, ..., TE_k][\nu]$ iff $\mathcal{M} \models A[TE_1] \wedge \cdots \wedge A[TE_k][\nu]$.

It follows from the semantics above that a tree expression $A[B[TE]]$ is equivalent to the formula $A[B] \wedge B[TE]$. Similarly, the tree expression $A[TE_1, \ldots, TE_k]$ is equivalent to the formula $A[TE_1] \wedge \cdots \wedge A[TE_k]$. As a concrete example of the former, $\texttt{Book} \rightarrow b[\texttt{publisher} \rightarrow p[\texttt{name} \rightarrow n]]$, says node b is labeled `Book`, has a child p labeled `publisher`, which in turn has a child n labeled `name`.

We next define the rules used for defining mappings. Rules are expressions of the form $\texttt{Head} \longleftarrow \texttt{Body}$, where `Head` is a tree expression and `Body` is a conjunction of tree expressions and interpreted predicates, such as $X > 5$, $X = Y$, etc. We assume as for Datalog that rules are range restricted and safe [45]. In particular, all variables appearing in `Head` appear in the `Body`. In view of the equivalences explained above, we can assume without loss of generality that `Head` is of the form A or $A[B]$, where A, B are atoms. More generally, we have the following:

Proposition 1 *Let* $\texttt{Head} \longleftarrow \texttt{Body}$ *be any range restricted safe rule in TreeLog. Let S be the set of atoms such that* `Head` *is equivalent to the conjunction of atoms in S. Then the above rule is equivalent to the set of rules* $\{A \longleftarrow \texttt{Body} \mid A \in S\}$.

The proof is trivial and is omitted. As an example, consider a rule of the form $A[B[C], D[E[F]]] \longleftarrow \texttt{Body}$. The semantics of a rule is that the body implies the head. Under this semantics, the above rule is equivalent to the set of rules:

$$A[B] \longleftarrow \texttt{Body},$$
$$A[D] \longleftarrow \texttt{Body},$$
$$B[C] \longleftarrow \texttt{Body},$$
$$D[E] \longleftarrow \texttt{Body},$$
$$E[F] \longleftarrow \texttt{Body}.$$

3.1.1 Tree transformations

TreeLog rules can express transformations over forests. We formalize this below. The notions of ground terms and atoms of Herbrand base are defined analogously to first-order logic. A tree expression is said to be ground if all atoms in it are ground. Let P be a set of rules in TreeLog, where each rule has a head of the form A or $A[B]$. Then the Herbrand base H_P associated with P is the set of ground atoms as well as ground tree expressions of the form $A[B]$. In what follows, we refer to a ground atom as well as a ground tree expression of the form $A[B]$ (where A and B are atoms) as a *fact*. Thus,

[4] We use upper case letters to denote variables.

the Herbrand base of P is the set of all ground facts formed using the symbols appearing in P.

P then defines a transformation $\mathcal{T}$ that maps subsets of H_P to subsets of H_P as follows. Let $S \subseteq H_P$. The set of ground facts implied by $P \cup S$ was denoted by $\tilde{\mathcal{T}}(S)$. Then we define $\mathcal{T}(S)$ to be the set of ground facts not in S that are implied by $P \cup S$, i.e., $\mathcal{T}(S) = \tilde{\mathcal{T}}(S) - S$.

Given any set of ground facts, G, we can obtain the graph represented by G as follows. (1) For each fact $t \rightarrow i$ appearing in G, create a unique node i with tag t. (2) For each fact $t \rightarrow i[t' \rightarrow i']$ in G, make i' a child of i.

For example, consider the following set of rules:

r_1: $\texttt{Publisher} \rightarrow f(P/\texttt{text}())[\texttt{name} \rightarrow P/\texttt{text}()] \longleftarrow$
$\texttt{Publisher} \rightarrow P$.

r_2: $\texttt{Publisher} \rightarrow f(P/\texttt{text}())[\texttt{Book} \rightarrow f(B)] \longleftarrow$
$\texttt{Book} \rightarrow B[\texttt{Publisher} \rightarrow P]$.

r_3: $\texttt{Book} \rightarrow f(B)[T \rightarrow f(N)] \longleftarrow$
$\texttt{Book} \rightarrow B[T \rightarrow N], T \neq \text{`}\texttt{Publisher}'$.

Consider an input forest consisting of books with subelements corresponding to title, authors, and publisher. Figure 4a and b show an example forest of book trees. Rule r_1 creates a new node $f(P/\texttt{text}())$ with label `Publisher`, for every publisher node P in the input. The node is created as a function of the data value of the input publisher node P. Thus, for every distinct publisher, one node is created. Further it creates a node labeled `name`, makes it a child of the publisher node created and associates the value $P/\texttt{text}()$ to this node. Rule r_2 creates a new node $f(B)$ for every input book node B. Whenever B has a publisher child P in the input, this rule makes $f(B)$ a child of the node $f(P/\texttt{text}())$. Finally, rule r_3, whenever the input contains a book node B with a child N whose label T is different than '`Publisher`', creates a node $f(N)$ and makes it a child of the book node $f(B)$ in the output. The transformation expressed by these rules is illustrated in Fig. 4c. It can be seen that the mapping rules above transform a forest of book trees into an output forest of publisher trees, where books are grouped by publisher.

A natural question is whether an arbitrary set of rules is guaranteed to transform a forest into a forest. This question is important since our approach for interoperability in HePToX is to derive mapping rules automatically from correspondences. Thus, we need the assurance that the derived rules always map forests to forests. As we show below, not all rules (or sets of rules) transform forests to forests. We give two examples. Consider the single rule $t \rightarrow i[t \rightarrow i] \longleftarrow$ (i.e., it has an empty body). This has the effect of making a node its own "child", effectively creating a cycle. As a less trivial example, revisit the set of rules $\{r_1, r_2, r_3\}$ above. Suppose in r_1, r_2, the term $f(P/\texttt{text}())$ is changed to $f(P)$ and the condition $T \neq \text{`}\texttt{Publisher}'$ is removed from the body of r_3. Then it is easy to see that when the rules are applied to an input forest of book trees, such as in Fig. 4a, then in the output, publisher nodes will be both a parent and child of book nodes, i.e., the output contains cycles.

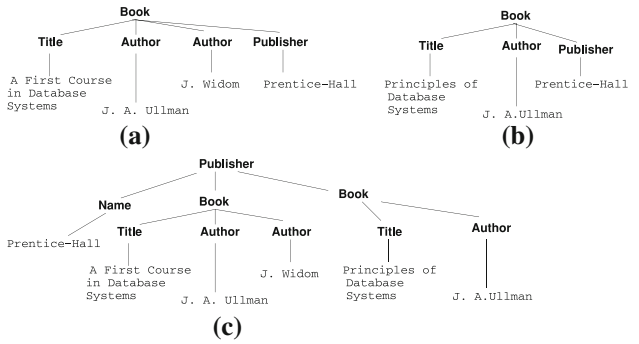


Fig. 4 An example of transformation from a forest of trees (**a**) and (**b**) into a forest (**c**)

Fortunately, a simple syntactic condition guarantees that a set of rules always transforms a forest into a forest:

Lemma 1 *Let P be a set of TreeLog rules of the form $T_1 \rightarrow I_1 [T_2 \rightarrow I_2] \longleftarrow \texttt{Body}$, such that the id term I_2 is a function of the id term I_1 and possibly other terms. Then P always transforms a forest into a forest.*

Proof First, notice that no self-loops will be ever created by P, since the id of a child is always a function of its parent, and thus is necessarily different from it. Suppose a node u in the output has two parents v_1, v_2. However, the id of u would have to be a function of v_1 and v_2, which is only possible if v_1, v_2 are the same node. Suppose the output contains a cycle $(v_1, v_2, \ldots, v_k, v_{k+1} = v_1)$. We have already shown the case $k = 1$ is impossible. A routine induction shows a cycle of any length which leads to a contradiction. □

Thus we have shown that HePToX's mapping language, TreeLog, is capable of supporting the complex relationships between schemas. Schema mapping systems, such as Clio [24,37] express mappings as source-to-target tuple-generating dependencies (s–t tgds), i.e., first-order formulas that formalize data exchange between a source and a target database. To enable the exchange, such dependencies are reformulated as executable mapping statements, i.e., in SQL or in XQuery. Our TreeLog mapping rules, although higher order, are semantically reducible to first order. In this sense, they are similar in spirit to the tgds employed in Clio. However, the higher order syntax of TreeLog permits it to express rules that map data elements to elements of schema and vice versa, unlike mappings expressed using s–t tgds. In addition, we use TreeLog mapping rules to enable query reformulation among heterogeneous peers, rather than data exchange, as highlighted before.

4 Inferring mapping rules from arrows

Given a pair of schemas, represented as graphs, and a set of arrows/boxes relating nodes across the graphs, HePToX must automatically infer a set of rules for mapping instances of one schema into instances of the other. In this section, we illustrate the HePToX automatic mapping rule inference algorithm, using the running example of transforming the representation in Fig. 2 into the rules in Fig. 3.

Suppose HePToX needs to infer mapping rules from a schema Δ_1 (call it source) to another Δ_2 (call it target) based on given correspondences (as in Fig. 2); the algorithm consists of the following main modules. (1) Determine groups of nodes in the two schemas such that each group intuitively captures some "unit" of information that should be considered separately. For example, consider the patient element in Fig. 1a. Since the hospital has many patients, and each patient has several characteristics, a "patient" is a good unit to consider as the origin or destination of a mapping rule. (2) For each source group, construct a tree expression describing the unit of information following the hierarchical structure of the group. Continuing our example: what information needs to be considered as part of the "patient" group? (3) For each target group, identify all minimal sets of source groups necessary to populate information into the target tree expression structure and construct the rules.

To appreciate the need for groups, consider mapping instances of Fig. 2a to those of Fig. 2b. Suppose we write a rule of the form: "⟨whatever⟩ ⟵ `MonGenHosp` → $Mon[`Patient` → $P[...], `Admission` → $A[...]]". Then we create the objects as per rule head, for each *combination* of patient and admission. Thus, the multiplicity of the elements in the original database will not be preserved by this rule. This problem will be solved if we write mappings for the following groups of nodes separately: {`MonGenHosp`, `Patient`, `@ID`, `MedCr#`, `Name`} and {`MonGenHosp`, `Admission`, `Problem`, `AdmDate`, `DisDate`, `@PatRef`}.

Sections 4.1, 4.2, and 4.3 show how HePToX automatically completes these steps. Section 4.4 shows how HePToX creates union mappings.

4.1 Identifying groups

As described earlier, the first step is defining which groups need to be mapped where. To do so, we first introduce the *group node*. The group node is the primary element under consideration, e.g., the "patient" in Fig. 1. Each group node induces a *primitive group*. This primitive group defines the characteristics that *must* be defined to describe the group node. Primitive groups model basic relationships that exist between different data items in a given schema and are thus similar to the primary paths of [37]. A main difference is that primitive groups account for nodes with boxes and thus facilitate mappings between data and schema. For example, the patient in Fig. 1 may be described as being a patient at MonGenHosp, and having an ID, MedCr#, and a Name.

$g_1 = \{MonGenHosp\}$
$g_2 = \{$MonGenHosp, *Patient*, Id, MedCr#, Name$\}$
$g_3 = \{$MonGenHosp, Patient, *Hist*, Event, Problem, Date$\}$
$g_4 = \{$MonGenHosp, Patient, *Treat*, Date, Desc, Doc$\}$
$g_5 = \{$MonGenHosp, Admission, *Admission_Problem*$\}$
$g_6 = \{$MonGenHosp, *Admission*, Admdate, Disdate, @Patref$\}$

Fig. 5 The primitive groups from `MON` as shown in Figure 2

The algorithm to find group nodes and primitive groups is described formally in Fig. 6, and informally works as follows. The root of a schema is a group node. Any node which has an incoming arrow labeled '?', '+', or '*', or is an ancestor of such a node is a group node. For example, in Fig. 1a, in the `MON` schema, all non-leaf nodes are group nodes. For the purpose of group formation, we turn the DAG into a tree by replicating nodes with multiple parents, if necessary. For example, in Fig. 1a, the `Problem` node would be replicated twice—once for `Admission` and once for `Event`. Note that the set of nodes inside a box is treated as a single node for the creation of group nodes. Let T be the resulting tree. Let u be a group node in T and suppose $v_1, \ldots, v_k$ are all its non-group children. Then the primitive group induced by u consists of u, all its ancestors in T, the children $v_1, \ldots, v_k$, and their descendants. The primitive groups corresponding to the `MON` schema of Fig. 1a are shown in Fig. 5. Note that since MonGenHosp has no non-group children, the group induced by it is just $\{MonGenHosp\}$.

To minimize mapping rules without losing expressiveness, primitive groups can be merged under some conditions, formalized below. Intuitively, these correspond to when two primitive groups are guaranteed to have the same cardinalities. Formally, let v be any node in the schema tree T above and u an ancestor of v. v is *mandatory* relative to u if no edge on the path from u to v is labeled '?' or '*'. Call v single-valued relative to u if no edge on the path from u to v is labeled '*' or '+'. Let g and h be two primitive groups. Let u be the least common ancestor of nodes in g and h. Then g and h can be merged provided: (i) all descendants of u in g are mandatory single-valued relative to the root; *or* (ii) all descendants of u in g are mandatory single-valued relative to u and all descendants of u in h are mandatory relative to u. In our example, we can merge g_1 with any one of the other primitive groups since condition (i) holds vacuously. Also, g_6 and g_5 have Admission as their least common ancestor, relative to which Admdate, Disdate, @Patref are mandatory single-valued, and Problem is mandatory. So, by (ii), g_6, g_5 can be merged. The resulting groups for the `MON` schema are shown in Figure 7, which also shows the groups for the `BOS`

Algorithm DetectGroups.
Input: a schema Δ.
Output: a set of groups of nodes from the graph of Δ.
1. Start from the root.
If there are multiple outgoing edges, mark the root as P;
Else follow the single path (no restriction on edge labels)
until it reaches a node that has multiple outgoing edges;
if there is none, return all nodes in Δ as one group;
else mark the node found as P;
2. Start from node P;
Follow each outgoing edge, and stop when we visit an edge
with label '$*$' or '$+$';
Mark the node that this edge points to as a "stop node";
If there are multiple edges with label '$*$' or '$+$'
pointing into nodes in a box
(e.g., Pulmonary, Coronary in Figure 2)
Mark the box as the stop node;
3. If there is at most one stop node found in Δ,
Group the descendants of current P node and all
the nodes on the path from root to P via previously
recursively marked P nodes (if any);
Else
For each stop node S {
If there are multiple outgoing edges,
mark this node as P;
Repeat (2) and (3) to find its stop nodes until it
reaches leaf nodes or find at most one stop node;
Else
Group the descendants of current stop node S,
and all the nodes on the path from root to S
via previous recursively marked P nodes; }
4. Group the rest of the nodes and their ancestor nodes together.

Fig. 6 Algorithm for detecting groups in schemas

(target) schema. *Note that groups always induce connected subgraphs of the schema graph.*

Figure 7 shows the groups created for the example in Fig. 2. HePToX's group detection algorithm (Fig. 6) handles disjunction in schemas. Disjunctions are treated as usual, by yielding alternative groups for each element which is or-ed in the disjunction. Since they do not affect the computation of groups, we do not discuss them further. Moreover, we do not consider cyclic schemas or ordered XML.

The group detection algorithm in Fig. 6 is applied to the `MON` schema in Fig. 1a as follows:

Step 1 The algorithm marks `MonGenHosp` as P (it has multiple outgoing edges).
Step 2 The algorithm visits `Patient` and (eventually) `Admi-ssion`, and marks them as stop nodes (since the edge labels are $*$). `Patient` is again (recursively) marked as P (>1 outgoing edge) and its associated stop nodes `Event` and `Treat` are found, each of which are again recursively marked as P.
Step 3 Since there are no stop nodes reachable from `Event`, the algorithm groups all descendants of `Event`, adds all nodes on the path from `MonGenHosp` (root) to `Event` and creates the group sg_1 in Fig. 7. The shared elements have been renamed (i.e., `Problem` and `Date` have become `EProblem` and `EDate`) to distinguish them from admission problem and admission date. In a similar fashion, the algorithm forms the groups sg_2 and sg_3.
Step 4 The algorithm forms the group sg_4, since `@ID, MedCr#, Name` are the left-over attributes and elements for the recursive call on `Patient` marked as a P node. Similarly, the algorithm will form tg_1, tg_2, tg_3 and tg_4 (as shown in Fig. 7) on the `BOS` schema in Fig. 2b.

Figure 7 shows the pairs of groups connected by arrows in our example. As will be shown in the remainder of this section, each target group corresponds to one mapping rule.

4.2 Generating tree expressions

At this point, HePToX knows what groups are formed based on the schema and how they are coupled based on the mappings. Having mapped the groups, the next step is to generate tree expressions—the building blocks of the mapping rules. For each group, g, we examine the subgraph of the original schema graph induced by the nodes in the group (not counting Key → KeyRef(s) edges). If the subgraph of the schema graph induced by g is a tree, we are ready to write the tree expression for that group. As in group formation, if the subgraph is a DAG, then we replicate each shared node (recursively) as many times as necessary to create a tree structure. An exception is when the DAG structure is the result of multiple nodes in a box which have the same substructure. For example, in the schema `BOS`, Pulmonary, Coronary, etc., are in the same box, and they have the same relationships with the same children. In this case, HePToX does *not* replicate the shared elements. Note that in our example, all source and target groups happen to induce trees.

For each source group, the tree expression is written by essentially following the recursive tree structure, using brackets—[]—to capture the nesting. For every node in the group, we write the expression `Tag` → $var, where `Tag` is the tag name of the node and $var is a new variable. For example, for group sg_1, we can write its tree expression as `MonGenHosp` → $`BOS`[`Patient` → $P[`Hist` → $H [`Event` → $E[`Pro−blem` → $EP, `Date` → $ED]]]]. If the node is inside a box (e.g., like Pulmonary in the `BOS` schema), then we use a tag variable and write $Tag → $var.

For target groups, HePToX follows the same procedure. However, there is a major difference w.r.t. source groups, i.e., we do *not* know the node ids needed in the generated tree expressions, since those have not yet been determined. Instead, we write the tree expression as a skeleton, leaving the node ids as '??' for now. As an example, the tree expression for tg_1 would be `BostonGeneral` → ?? [`Admission` → ?? [$Tag → ?? [`@ID` → ?? , . . . , `Patient` → ??]]]. The '??' will be filled in when we write the mapping rules in Sect. 4.3. We drop a leaf node from consideration if there

Groups created by the algorithm in Figure 6.
Each target group is numbered with the rule number that it corresponds to in Figure 3. Group nodes are in italics.

Source groups:
sg_1 = {MonGenHosp, Patient, *Hist*, Event, EProblem, EDate} //corresponds to primitive group g3
sg_2 = {MonGenHosp, Patient, *Treat*, TDate, EDate, TDesc, Doc} //corresponds to primitive group g4
sg_3 = {MonGenHosp, *Admission*, *Admission_Problem*, AdmDate DisDate, @PatRef}//corresponds to primitive groups g5 and g6
sg_4 = {MonGenHosp, *Patient*, @ID, MedCr#, Name}
//corresponds to primitive groups g2 and g1 (though note that g1 could have been merged with any other group
Target groups:
tg_1 = {BostonGeneral, Admission, Pulmonary, Coronary, @ID, InsName, Policy#, Enter, Leave, Patient}
tg_2 = {BostonGeneral, Progress, Symptom, SDate, SDesc}
tg_3 = {BostonGeneral, Progress, Treatment, TDate, TDesc}
tg_4 = {BostonGeneral, Progress, @PatRef}.
Pairs of Groups Connected by Arrows:
$sg_3 \leftarrow tg_1$, $sg_4 \leftarrow tg_1$, $sg_1 \leftarrow tg_2$, $sg_2 \leftarrow tg_3$, $sg_3 \leftarrow tg_4$.

Fig. 7 Example of group determination for Fig. 2

Algorithm DeriveMappingRules.
Input: source groups, and target groups
Output: a set of mapping rules
For every target group tg
Let $\{sg_i, ..., sg_j\}$ be the set of source groups connected to tg by arrows.
Let $TE(tg)$ be the tree expression for group tg.
1.Start with the rule skeleton $TE(tg) \longleftarrow TE(sg_i), ..., TE(sg_j)$
Fill in the variables corresponding to leaf positions in $TE(tg)$ based on the arrows incident on the leaf elements of tg
2.For root and each of its descendants via only single-valued edges
Assign their ids as distinct Skolem functions of the root variable in the source
3.For each internal node $inode$
Assign its id as a distinct Skolem function of the variables associated with all its single-valued leaf descendants
If any of its single-valued leaf descendant sc does not belong to tg
Trace the source group sg_p with an arrow pointing to sc
Add $TE(sg_p)$ in the rule body: $TE(tg) \longleftarrow TE(sg_i), ..., TE(sg_j), TE(sg_p)$
Let $scref$ denote the corresponding element of sc in the source schema
If $scref$ points to a node in the source schema $scid$ and the source group sg_q that $scid$ belongs to is not in $\{sg_i, ..., sg_j\}$
Add $TE(sg_q)$ in the rule body: $TE(tg) \longleftarrow TE(sg_i), ..., TE(sg_j), TE(sg_p), TE(sg_q)$
Equate the variable $scref$ binds to ($\$scref$) with the variable $scid$ binds to ($\$scid$): $\$scref$=$\$scid/text()$
(if $scid$ is an attribute, then use $\$scref$=$\$scid$ instead)
Add $\$scref$ as an argument to the Skolem function on the RHS of $inode$.

Fig. 8 Rule construction algorithm

is no counterpart in the source schema. For example, Doc is one such node. The module for creating trees and for writing tree expressions is straightforward and is omitted for brevity.

4.3 Generating mapping rules

At this point, the groups that should be mapped together in a single mapping rule have been decided, and the basic format of each rule has been decided. The last step is writing the mapping rules. Figure 8 shows the formal algorithm specification. To make the algorithm clearer, we now show how to create the rule for $TE(tg2) \longleftarrow TE(sg1)$, which is Rule 2 in Fig. 3. Consider each target group tg. Let $\{sg_i, \ldots, sg_j\}$ be the set of source groups connected by arrows to tg. Let $TE(g)$ denote the tree expression for group g. Applying step 1, we start with the rule skeleton $TE(tg) \longleftarrow TE(sg_i), \ldots, TE(sg_j)$. Based on the arrows incident on the leaf elements of tg, we fill in the variables corresponding to leaf positions in $TE(tg)$, i.e., the right-hand-side of atoms corresponding to leaf nodes. For example, for tg_2, we start with $TE(tg_2) \longleftarrow TE(sg_1)$. The rule body only contains $TE(sg_1)$ since that is the only source group connected to tg_2 by arrows (see Fig. 7). Based on the arrows, we can fill in the right-hand-sides of Date and Desc in the rule head as $\$ED$ and $\$EP$, respectively:

BostonGeneral → ?? [Progress → ?? [Symptom → ?? [Date → ??, Desc → ??]]] ⟵ MonGenHosp → $\$M$ [Patient → $\$P$ [Hist → $\$H$ [Event → $\$E$ [EProblem → $\$EP$, EDate → $\$ED$]]]].

Next, according to step 2, we assign as IDs for the root BostonGeneral and its single-valued child Admission distinct Skolem functions of the root variable in ($BOS).

Next, step 3 creates ids for each internal node. HePToX uses a key as an ID if one is provided by the schema. Otherwise, it is necessary to construct a key. Specifically,

HePToX uses a distinct Skolem function of the variables associated with all its mandatory single-valued leaf descendants. Additionally, if a variable is used for the tag of v, then this variable is also added as a Skolem argument. For example, for the RHS of `Symptom`, the key is the Skolem function $f3(\$EP, \$ED)$, since no key is specified. For `Progress`, its (only) mandatory single-valued leaf child is `@PatRef`, which does not belong to tg_2. By following the arrow incident on `@PatRef`, we trace source group sg_3, so we introduce $TE(sg_3)$ in the rule body. Now `@Pat-Ref` in the source schema points to the ID attribute `@ID` of `Patient`, an attribute that does not belong to sg_1 or sg_3. However, `@ID` belongs to sg_4, so we also add $TE(sg_4)$ to the above rule body. We equate the variable associated with the `@ID` attribute in $TE(sg_4)$ with the variable associated with `@Pat-Ref` in sg_1. At this point, the rule is as follows:

$$
\begin{aligned}
&\texttt{BostonGeneral} \rightarrow f1(\$\texttt{BOS})\ [\texttt{Progress} \rightarrow ?? \\
&\quad [\texttt{Symptom} \rightarrow f3(\$ED, \$EP)\ [\texttt{Date} \rightarrow \$ED, \texttt{Desc} \rightarrow \$EP]]] \\
&\qquad\qquad \longleftarrow \\
&\quad \texttt{MonGenHosp} \rightarrow \$M\ [\texttt{Patient} \rightarrow \$P \\
&\qquad [\texttt{Hist} \rightarrow \$H\ [\texttt{Event} \rightarrow \$E\ [\texttt{Problem} \rightarrow \$EP,\ \ \texttt{Date} \rightarrow \$ED]]]], \\
&\quad \texttt{MonGenHosp} \rightarrow \$M\ [\texttt{Admission} \rightarrow \$A\ [\texttt{Problem} \rightarrow \$AP, \\
&\qquad \texttt{AdmDate} \rightarrow \$AD, \texttt{DisDate} \rightarrow \$DD, \texttt{@PatRef} \rightarrow \$PR]], \\
&\quad \texttt{MonGenHosp} \rightarrow \$M\ [\texttt{Patient} \rightarrow \$P\ [\texttt{@ID} \rightarrow \$I, \texttt{Name} \rightarrow \$N, \\
&\qquad \texttt{MedCr\#} \rightarrow \$MC]], \$PR = \$I.
\end{aligned}
$$

For the RHS of `Progress` we can assign the Skolem function $f2(\$PR)$. We refine the rule by identifying nodes/paths shared between two or more tree expressions in the body. This yields rule 2 in Fig. 2. Note that the generated rules are always *safe*—all variables in the head appear in the body.

4.4 Extending HePToX to union mappings

HePToX supports union mappings. In a PDMS scenario, unions may be expressed according to two different semantics, depending on the relationships between labels; e.g., Fig. 9a shows correspondences between schemas *src* and *tgt*. Here, *src* consists of a set of r-tuples and a set of s-tuples, while *tgt* consists of a set of t-tuples. We need to map the union of r and s in *src* to t in *tgt*. Figure 9a shows two possible alternatives. In scenario 1, r and s correspond to vehicle and car (perhaps *src* was created by integrating two other schemas) and t corresponds to automobile. Intuitively, vehicle $\cup$ car $=$ automobile, thus the union of the set of r-tuples and the set of s-tuples is equal to the set of t-tuples. Note that all of them model the make and price of vehicles. In scenario 2, r corresponds to European cars (euroCar) while s to American cars (amCar). In this case, we expect some kind of subsumption relationship to hold, i.e., euroCar $\cup$ amCar $\sqsubseteq$ automobile. To allow both kinds of unions, we propose the use of a *union mapping table*. A union mapping table is very similar to a mapping table [5]. It is a table with three columns (*source schema labels*, *op*, and *target schema labels*). Each row of this table states the correspondence between the union of source schema labels (tags) and a target schema label using one of the operators $=, \sqsubseteq$.

Unions represent 1–m mappings, and, as such, can be extended to represents m–n mappings in HePToX. By widening the example in Fig. 9a, an euroCar element in the source can be connected via union with containment to both a localCar element and a foreignCar element in the target, i.e., euroCar $\cup$ amCar $\sqsubseteq$ localCar $\cup$ foreignCar in Table 1. Figure 9b shows the new target schema and possible instances in the source database and in the target database.

Notice that the semantics of m–n mappings, as a generalization of 1–m union mappings, say that the same value (FIAT) can be a localCar, e.g., with a the target schema belonging to an Italian peer, or, alternatively, can be a foreignCar, e.g., with a target schema belonging to a Spanish peer. If the containment holds, the target instance can also include further values (e.g., Toyota). M–n unions with equality hold in a similar fashion, by imposing that euroCar $\cup$ amCar $=$ localCar $\cup$ foreignCar (cfr. last row in Table 1).

4.5 Final remarks

The main contribution of this section has been the automatic inference of mapping rules that transform tree database instances of one schema into those of another. We have studied how to determine groups of information and used them to create the mapping expressions, which constitute the components of a rule. Moreover, we have discussed an extension of HePToX to handle union mappings. Incidentally, our group detection algorithm is reminiscent of the tableaux formation in Clio mapping generation algorithm [24,37]. However, the rest of the mapping generation algorithm is unique to HePToX, as it is guided by the higher order syntax of TreeLog and by data-metadata correspondences.

To conclude this section, we next briefly address the question, what is significant or fundamental about the class of transformations that are captured by the rules? To avoid detracting from the main point of the text, we refer the reader to the appendix [22].

The key idea illustrated in the appendix is that the rules capture a class of database tree transformations that are expressible using the operators unnest/nest (similar to those for nested relations), flip/flop (which basically change nesting orders in the schema), and merge/split (which have a flavor of grouping and "ungrouping" a set of nodes). It can be shown that the rules capture precisely the class of transformations expressible using these operators together with a few additional operators like node addition/deletion and tag modification, added for "completeness" purposes.

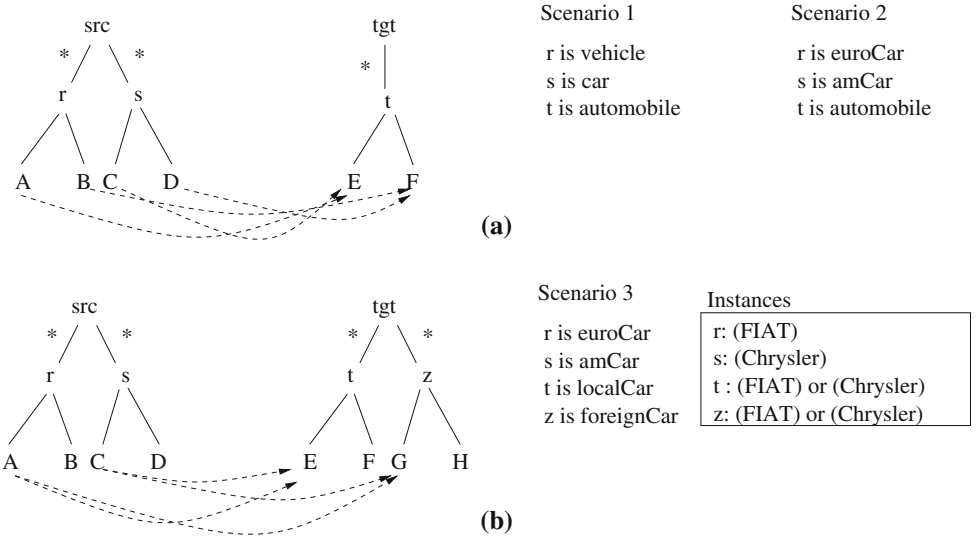


Fig. 9 **a** Unions in scenarios 1 and 2; **b** m–n Mappings in scenario 3. $A = C = E = G = make$ and $B = D = F = H = price$

5 Query translation

This section addresses two questions: (1) Suppose there are a pair of peer XML database sources p_i, with DTDs Δ_i and underlying database instances D_i, $i = 1, 2$. Suppose a query Q is issued against the DTD of p_1 (p_2). What does it mean for Q to be answered using the database of p_2 (p_1)? (2) Can Q be translated into a new query, Q', such that (a) Q' is over the other peer's DTD and (b) $Q'(D_2)$ yields the correct answers w.r.t. the semantics captured by the answer to question (1)? Further, can this translation be done efficiently?

5.1 Query translation semantics

Suppose mapping rules μ map instances of schema Δ_1 of one peer to those of schema Δ_2 of another peer, i.e., $\mu : \Delta_1 \rightarrow \Delta_2$; e.g., Fig. 3 shows `MON` $\rightarrow$ `BOS`. Let $inst(\Delta)$ denote the set of instances of Δ. We begin by discussing query translation semantics where only 1–1 mappings are considered; we talk about extending this to union mappings in Sect. 4.4.

Definition 1 (*Semantics*) Suppose Q_i is a query posed against $\Delta_i, i = 1, 2$. Let Q_i^t denote a translation of Q_i against Δ_j, $j \neq i$. Then Q_2^t is *correct* provided $\forall D_1 \in inst(\Delta_1) : Q_2^t(D_1) = Q_2(\mu(D_1))$. The translation Q_1^t is *correct* provided $\forall D_2 \in inst(\Delta_2) : \; Q_1^t(D_2) = \bigcap_{D_1^k : \mu(D_1^k) = D_2} Q_1(D_1^k))$.

In other words, the translation Q_2^t is correct provided Q_2 applied to the transformed instance $\mu(D_1)$ and Q_2^t applied to D_1 both yield the same results, for all $D_1 \in inst(\Delta_1)$. Note that in this case, the direction of translation is *against* that of the mapping μ. We henceforth call this *backward query translation*. Translating a query Q_1 posed against Δ_1 to the schema Δ_2 of peer p_2 is *aligned* with the direction of the mapping μ. We call this direction *forward translation*. Intuitively, backward translation is similar to view expansion and is the easier of the two. A key complication in forward translation is that μ, the mapping that transforms instances of Δ_1 to those of Δ_2, may not be invertible [17]. Thus, we define the semantics of query answering based on certain answers over all possible pre-images D_1^k for which $D_2 = \mu(D_1^k)$. We begin by describing forward translation in Sect. 5.2. Section 5.3 briefly describes backward translation. Section 5.4 discusses the correctness of HePToX's query translation algorithms.

Output formatting: Note that owing to schema discrepancies between Δ_1 and Δ_2, the output of $Q_2(\mu(D_1))$ would adhere to the schema Δ_2. For $Q_2^t(D_1)$, XQuery permits restructuring of output, so this is not an issue. A similar comment applies to the translation Q_1^t of Q_1.

5.2 Forward query translation algorithm

Given a mapping μ from DTD Δ_1 to DTD Δ_2 ($\mu : \Delta_1 \rightarrow \Delta_2$), we give a query translation algorithm that seamlessly works when the input query Q_i is posed against either Δ_1 or Δ_2. The forward direction of query translation corresponds to answering queries using views. In this paragraph, we illustrate the forward direction.

XQuery fragment considered: The fragment of XQuery we consider corresponds to queries expressible as joins of tree patterns (TP) (see [3]), where the return arguments correspond to leaf nodes of the database. Even for this simple fragment of XQuery, query translation is far from trivial.

5.2.1 Translating tree patterns

We use the following XQuery, Q_1, on the MON schema (Fig. 1a) as a running example to illustrate forward translation: "Find all patients with Admission/Problem = 'Coronary' whose Treatment started Dec 25, 2003", expressed as:

Example 1 [XQuery Forward]

```
Q1: FOR $A IN //Admission,
        $P IN //Patient[@ID=$A/@PatRef]
WHERE $A/Problem="Coronary" AND
        $P/Treat/Date="12/25/2003"
RETURN {$P/Name}
```

We represent this query as (a join of) two tree patterns, as illustrated in Fig. 10.

The rest of this section focuses on translating single tree patterns. Translation of joins of TPs is discussed in Sect. 5.2.2. Our algorithm translates a tree pattern for each relevant mapping rule by applying two main steps: (i) *expansion*, and (ii) *translation*. After the tree patterns have been translated w.r.t. all relevant mapping rules, they undergo (iii) a *stitching* phase and possibly (iv) a *contraction* phase. We detail each phase in the following.

Expansion: The first forward query translation step is expanding a query—which is represented by a tree pattern—to the body of a rule. The goal is to match the TP and the rule body so that the TP can be mapped to the rule's body. Formally, let t be a TP and $r : h_r \longleftarrow b_r$ be a mapping rule, where h_r is the rule head, and b_r is the rule body. Expansion helps match t to b_r, which is a substitution θ that maps variables in b_r to those in t. This substitution θ may be partial since b_r may contain components which have no counterpart in t, and not all nodes in t may be in the range of θ. Any nodes that appear in the rule body but do *not* have a TP variable mapped to them are called *dummy nodes*. If we find a non-empty substitution θ, check whether any variable $\$X$ in b_r such that $\theta(\$X)$ is a leaf variable of t, appears in the rule head h_r. If not, then r is not *relevant* for translating t.

For example, Fig. 10, our running example, shows (the join of) two TPs—t_1 and t_2. Consider the mapping rules in Fig. 3 again. Figure 11a shows t_1 without its join condition. Figure 11b shows the expansion of the body of Rule 1 in Fig. 3. The part of the expanded TP that was originally present in t (Fig. 11a) is highlighted in Fig. 11(b) by using nodes shown in dark circles (e.g., $A, $AP, and $PR). To better illustrate the dummy nodes—those nodes not associated with variables in Fig. 11a—in Fig. 11b (e.g., $AD, $M, and $P, etc) have edges leading to them shown as dashed gray lines. At this point, any distinguished node present in the original TP is also identified as such in the expanded TP and distinguished nodes are tracked as so through the steps of the query translation algorithm. Therefore, Rule 1 is relevant for tree pattern t_1.

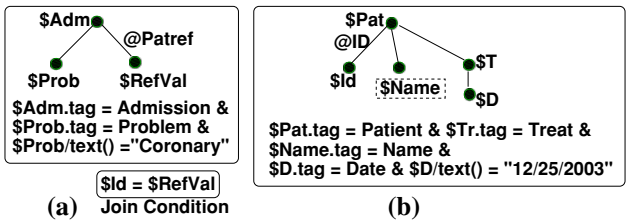


Fig. 10 'Join of' two tree patterns

Similarly, Rules 3 and 4 are relevant for t_1, while rules 1 and 3 are relevant for t_2. For instance, rule 2 is irrelevant for both TPs since no variables corresponding to the TP variables (via any substitution) appear in the head of rule 2.

At this point, any distinguished node present in the original TP is also identified as such in the expanded TP, and distinguished nodes are tracked as such throughout the steps of the query translation algorithm. Since t_1 has no distinguished nodes, there are no changes to track in Fig. 11.

Translation: Next, we translate the expanded TP by applying the rule to it. The correspondence between the variables in the original TP and those in the expanded TP (i.e., the rule body) is kept track of by means of a substitution between the two. Figure 11b shows the substitution for TP t_1 as $\{\$A \rightarrow \$Adm, \$AP \rightarrow \$Prob, \$PR \rightarrow \$RefVal\}$. Using this substitution, original query constraints are propagated through the translation; e.g., the translated query resulting from applying rule 1 to the expanded TP above is shown in Fig. 11c. For readability, all tag constraints are shown concisely by writing the tags (in gray) next to the appropriate nodes. Note how the constraint $\$Prob/text() = "Coronary"$ is propagated via the substitution as $\$AP/text() = "Coronary"$. Additionally, the condition $\$PR = \$ID$ in the body of rule 1 is used to infer that the attribute child $@ID$ of the node $\$AP/text()$ in Fig. 11c corresponds to the attribute child $@PatRef$ of the node $\$A$ in Fig. 11b. Note that some of the nodes have Skolem terms associated with them. They play a key role in the stitching phase.

Figure 11d–e shows the translated query pieces obtained from t_1 via rules 3 and 4, respectively. Figure 12a reproduces the TP t_2 while Fig. 12b–d shows its translated query

Table 1 Union mapping table in HePToX

Source elements	*Operator*	*Target elements*
vehicle ∪ car	=	automobile
euroCar ∪ amCar	⊑	automobile
vehicle ∪ car	=	localCar ∪ foreignCar
euroCar ∪ amCar	⊑	localCar ∪ foreignCar

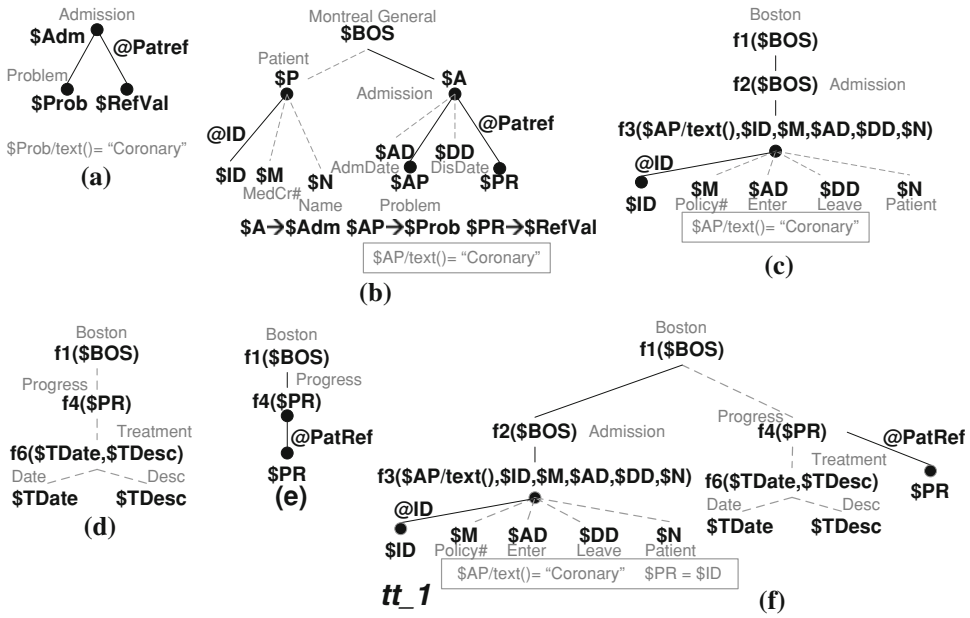


Fig. 11 A series of steps in the translation of TPs t_1 from Fig. 10: **a** Tree pattern t_1 from Fig. 10a without the join predicate. **b** The expansion of the body of Rule 1 from Fig. 3 to the tree pattern t_1. **c** The results of applying the head of Rule 1 to the expansion of t_1 (as shown in (**b**)). **d** The results of applying the head of Rule 3 to the expansion of t_1. **e** The results of applying the head of Rule 4 to the expansion of t_1. **f** The results of stitching together the translations in (**c**), (**d**), and (**e**)

pieces obtained via rules 1, 3, and 4. Distinguished nodes are shown with a box surrounding their variable (e.g., $\$N$, $\$N2$, in Fig. 12).

The last part of the translation phase is to remove the dummy nodes from the *leaves* of the translated query, since they are not necessary in the translation of the query. We cannot remove the dummy nodes from *internal* nodes, since they may be used for stitching. For example $\$M$, $\$AD$, $\$DD$ can be dropped in Fig. 11c. In contrast, internal dummy nodes cannot be dropped at this stage, and they are actually kept as such in the translated TP of Fig. 11c. Examples of such nodes are $f1(\$\texttt{BOS})$, $f2(\$\texttt{BOS})$, $f3(\$AP/text())$, $\$ID$, $\$M$, $\$AD$, $\$DD$, $\$N$), which are Skolem functions of dummy nodes in the expanded TP. The latter may indeed be needed later during the stitching phase.

Stitching: At this stage we have obtained translated pieces of a TP via various mapping rules. They must be stitched together by identifying nodes and possibly adding equalities between leaf variables. Two nodes are identifiable provided they have the same tag and the Skolem terms denoting their node id are unifiable. Consider the translated pieces associated with t_1 (Fig. 11c–e). It is easy to see that the two `BostonGeneral` nodes and the two `Progress` nodes are both identifiable: unification is via an identity substitution. Stitching those trees based on this identification yields the TP in Fig. 11f. The key/keyref constraint between `@ID` and `PatRef` is used to infer the equality $\$PR = \$ID$, which is added as a condition in Fig. 11f.

Figure 12b–d shows the translated pieces of t_2 (Fig. 12a) w.r.t. Rules 1, 3, and 4. The result of stitching is shown

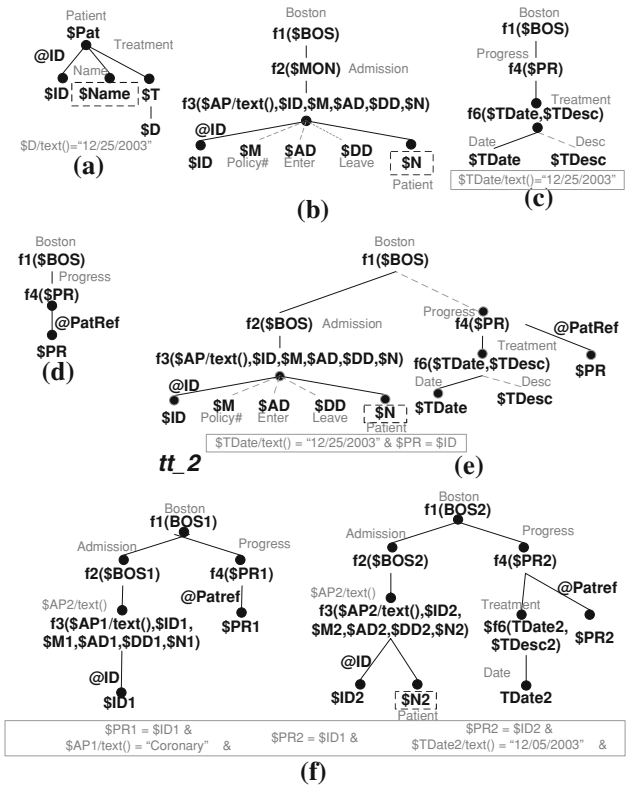


Fig. 12 Translation of TP t_2 from Fig. 10: **a** TP t_2 from Fig. 10b without the join predicate. **b** The results of applying the head of Rule 1 to the expansion of t_2. **c** The results of applying the head of Rule 3 to the expansion of t_2. **d** The results of applying the head of Rule 4 to the expansion of t_2. **e** Stitching together the results of (**b**), (**c**), and (**d**). **f** The two individual translated TPs, tt_1 (corresponding to t_1) and tt_2 (corresponding to t_2)

in Fig. 12e. Three BostonGe-neral nodes and the two Progress nodes have been identified and the equality $\$PR = \$ID$ has been added to the conditions.

Contraction: This step drops dummy nodes from the translated query. A node in a translated TP is a dummy node provided it (1) is a leaf and corresponds to a dummy node in the original expanded source TP, or (2) it is an internal node and all its children are dummy. In Fig. 11f, the dummy nodes (e.g., $\$M$, $\$TDate$, $\$TDesc$, $f6(\$TDate, \$TDesc)$) are highlighted by graying out the edges leading to them. Note that $f6(\$TDate, \$TDesc)$ is dummy since both its children are dummy. The translated TP corresponding to t_1 is simply the TP in Fig. 11f with all dummy nodes dropped and all Skolem terms dropped and replaced by the tags associated with those nodes. Similarly, $\$M$, $\$AD$, $\$DD$, and $\$TDesc$ in Fig. 12e are dummy nodes. The translated TP corresponding to t_2 is just the TP in Fig. 12e with the dummy nodes dropped and the Skolem terms dropped and replaced by the tags associated with those nodes.

5.2.2 Translating tree patterns joins

Consider again the XQuery query Q1 and its representation as join of TPs in Fig. 10. This query is translated by first translating each of the TPs and then adding in the join condition. We rename the variables separated across translated TPs to avoid conflict. Figure 12f shows the two individual translated TPs (denoted tt_1 and tt_2) corresponding to t_1 and t_2 along with the join condition. The original join condition was $\$PR = \$ID$, which, after variable renaming, becomes $\$PR2 = \$ID1$. Note that the dummy nodes present in the translations of t_1 (Fig. 11f) and t_2 (Fig. 12e) are dropped in Fig. 12f. To get the final translated query, we need to detect nodes across the two tree patterns tt_1 and tt_2 that need to be merged. This can happen because of constraints in the query. For example, the join condition $\$PR2 = \$ID1$, together with the equality $\$PR2 = \$ID2$, implies $\$ID1 = \$ID2$. Since the node $\$ID1$ corresponds to a key (of Coronary, Pulmonary, etc.) according to the BOSschema, this means that the two @ID nodes in tt_1 and tt_2 whose values have been equated, must be identical as nodes. Since the database is tree-structured, the parents $f3(\$AP1/text(), \$ID1, \$M1, \$AD1, \$DD1, \$N1)$ and $f3(\$AP2/text(), \$ID2, \$M2, \$AD2, \$DD2, \$N2)$ of the two @ID nodes should also be the same. This induces the constraint $\$AP1/text() = \$AP2/text()$ and hence $\$AP2/text() =$ 'Coronary'. Similarly, the equalities $\$PR2 = \$ID1$ and $\$ID1 = \$PR1$ imply $\$PR2 = \$PR1$ and hence the two Progress nodes in Fig. 11f with node id $f4(\$PR1)$ and $f4(\$PR2)$ are identified, and hence the two @PatRef nodes, being single-valued children of their parent, are identified in turn. In general, whenever two nodes are identified, so are their parents in the two trees. This process is shown in Fig. 13a. The final merged TP with Skolem terms dropped is shown in Fig. 13b. The XQuery corresponding to this TP is as follows:

```
Q1: FOR $C IN /BostonGeneral/Admission/Coronary,
        $P IN /BostonGeneral/Progress
WHERE $C/@ID=$P/@PatRef AND
        $P/Treatment/Date="12/25/2003"
RETURN {$C/Patient}
```

5.3 Backward query translation algorithm

When a query Q is expressed against the DTD Δ_2, the key intuition for query translation is to follow the mapping rule in the reverse direction, i.e., from the head to the body. This has resemblances to query folding and answering queries using views in [29]. However, the presence of Skolem functions greatly simplifies this process. The reason is that the node ids act as a "glue" suggesting which sub-element pieces should be associated together. Consequently, they drive exactly which mapping rule bodies we need to "join" together to rewrite the given query. A backward query translation algorithm has been developed in [46], with nested queries and target constraints. For a thorough discussion between our backward query translation algorithm and theirs, see Sect. 7. We now explore this algorithm through an example; the pseudocode is shown in Fig. 16.

Example 2 **[XQuery Backward]** Consider the query: "Find symptoms of patients admitted with 'Pulmonary' (ailment)".

```
Q2: FOR $P IN //Progress,
        $Y IN //Pulmonary
WHERE $Y/@ID=$P/@PatRef
RETURN {$P/Symptom/Desc}
```

Figure 14 shows the join of TPs corresponding to this query. Next, HePToX translates each query TP by matching it to each rule head. While similar to the expansion phase in Forward Query Translation, here the query TPs are matched to the *head* of each rule, rather than the *body* of each rule.

Continuing with our example, it is easy to see that only the heads of 1, 2, 4 can be partially matched to the query.[5] The expanded TPs are obtained by matching the TPs against each of these rule heads and are shown in Fig. 15a–c, where to minimize clutter, we do not show tag constraints nor the substitutions between query variables and the terms in the rule heads. From the names of variables and tags the substitution

[5] In the head of rule 3, only Progress can be matched to the query, but this is subsumed by matches to other rule heads and is redundant.

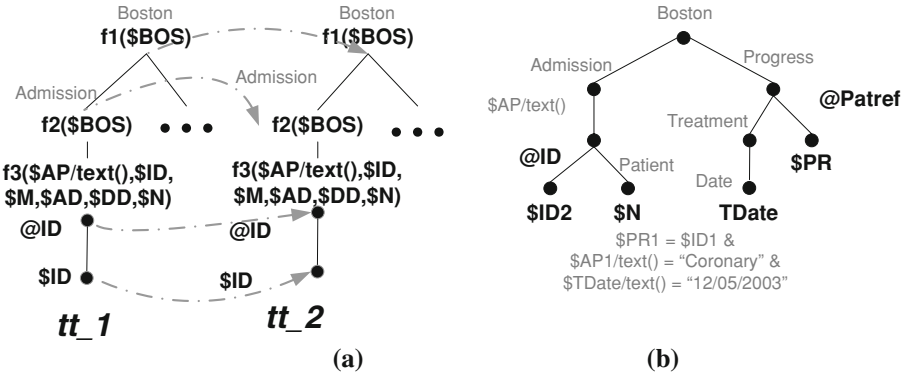


Fig. 13 Translation of the join of TPs from Fig. 10: **a** some pair of nodes in tt_1 and tt_2 being merged; **b** the final TP

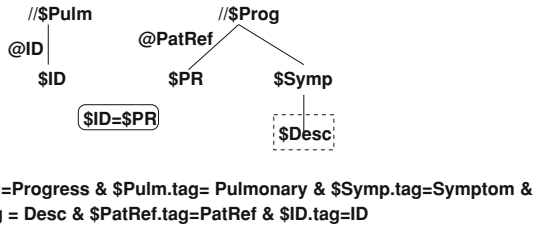


Fig. 14 Join of TPs corresponding to Q2

between query variables and rule variables should be implicitly clear. We also mark the variables that appeared in the original query (shown via distinctly patterned edges in the figure). For example, in Fig. 15a–c, we know \$Pulm is associated with the node $f_3(\$AP/text(), \$ID, \$M, \$N, \$AD, \$DD)$, \$ID with the node \$ID, \$Prog with node $f_2(\$PR)$, \$Symp with node $f_3(\$EP, \$ED)$, \$Desc with node \$Desc, \$Prog also with the node $f_2(\$ID)$, and finally \$PR with node \$PR.

Next, each expanded TP is replaced by the tree expression in the corresponding rule body (Fig. 15d–f). We again track the variables mentioned in the original query. For example, query variable \$Pulm corresponds to each of the nodes labeled \$AP in Fig. 15d–f and query variable \$Desc corresponds to the node labeled \$EP in Fig. 15e. Other variables are tracked in a similar fashion.

The next step is to drop dummy nodes. The idea is very similar to that in the forward direction of translation and is not elaborated further. Dropping of dummy nodes generates simplified but equivalent TPs. Nodes in different TPs that correspond to the same query variable are stitched together. For example, the \$A and \$PR nodes in Fig. 15e and f are merged. The final result of merging nodes is shown in Fig. 15g, which is a join of two TPs. The TPs are shown more concisely by writing the tags directly in place of the variables that are constrained by those tags. The join of TPs corresponds to the following XQuery statement:

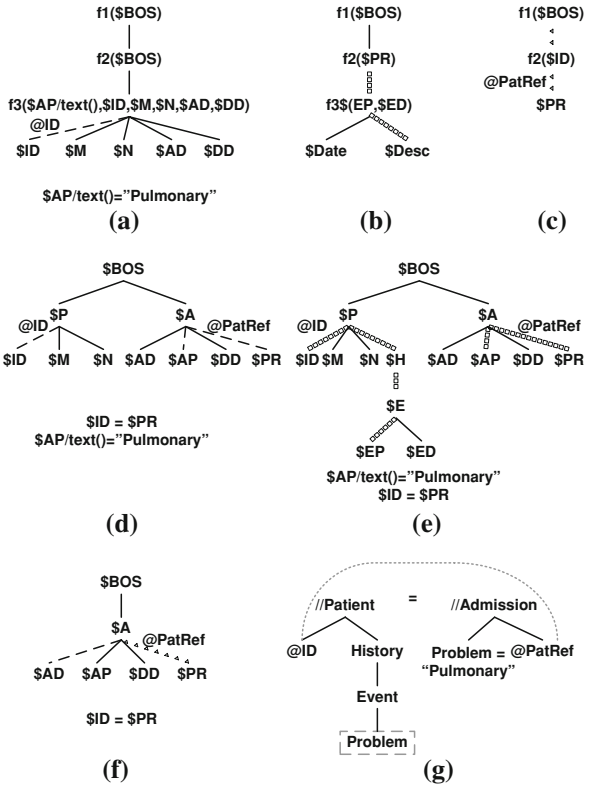


Fig. 15 Translating $Q2$ with the rules in Fig. 3: **a** Matching the query patterns for $Q2$ with the head of Rule 1. **b** Matching the query patterns for $Q2$ with the head of Rule 2. **c** Matching the query patterns for $Q2$ with the head of Rule 4. **d** Replacing the head pattern of (**a**) with the body of Rule 1. **e** Replacing the head pattern of (**b**) with the body of Rule 2. **f** Replacing the head pattern of (**c**) with the body of Rule 4. **g** The final result

```
FOR $P IN //Patient,
    $A IN //Admission[@ID=$P/@ID]
WHERE $A/Problem='Pulmonary'
RETURN {$P/Event}
```

5.4 Query translation correctness

Figure 16 shows forward direction query translation algorithm. The algorithm translates one TP at a time. For each

rule which is relevant to a given TP, it expands the TP and translates it. Each rule may only translate a piece of the TP in general, since the matching between the TP and the rule's body may be partial. The algorithm then stitches the translated pieces obtained via various rules using the stitching procedure explained above. It then joins the translated TPs. This "join step" involves variable renaming, a chase procedure for identifying nodes across the translated TPs, and replaces the Skolem terms by the tags associated with the nodes. Finally, the resulting TP (or join of TPs) is translated to XQuery, a step that is straightforward and is omitted.

The following result shows the correctness of the algorithm w.r.t. the query answering semantics in Definition 1.

Theorem 1 [Correctness of query translation] *For the fragment of XQuery defined in Sect. 5.2, the query translation algorithm in Fig. 16 is correct w.r.t. the semantics of query answering defined in Definition 1.*

Proof To prove the correctness of query translation, we need to show that it holds for both forward and backward directions of query translation.

We first show that it holds for the backward direction and use some of the results for the converse translation. The claim for this direction can be formulated as follows:

Let μ, Δ_1, Δ_2, Q_2, Q_2^t be as defined above. Then, for every $D_1 \in inst(\Delta_1)$: $Q_2^t(D_1) = Q_2(\mu(D_1))$.

The backward translation (also reported in Fig. 16) is similar to view expansion, with μ being the view definition. Intuitively, the composition $\mu \circ Q_2$ should give us Q_2^t. This is a sanity check that we must run, and the Q_2^t we get using our algorithm must be equivalent to this.

Let ι: $\mathcal{V} \mapsto \mathcal{D}$ be a variable instantiation as a function that maps each variable $\in \mathcal{V}$ in the query (whether it be Q_2 or Q_2^t) to a value in a database D (which may be $\mu(D_1)$ or D_1). Thus, given a query Q, containing a set of variables $\{\$x_1, \ldots, \$x_n\}$, an instantiation ι maps each variable $\$x_i$ to a value in a given database. Thus, an instantiation can be seen as a tuple of bindings over $(\$x_1, \ldots, \$x_n)$.

An instantiation ι is valid provided this tuple satisfies any applicable predicate condition (e.g., $price > 100, \$idref = \$id$, etc.). More precisely, let ι be a variable instantiation for query Q_2^t over D_1. Let ι' be a variable instantiation for query Q_2 over $\mu(D_1)$, ι' is obtained via a constructing procedure as $\iota \circ \mu^{-1}$. Indeed, ι is valid as it satisfies any applicable predicate condition p. μ^{-1} is valid provided it satisfies any applicable predicate p^t (with p^t being the predicate translated according to μ^{-1}), plus the other predicates possibly introduced by μ^{-1} and due to joins of variables in the rule body. Given that ι is valid, and μ^{-1} is valid, ι' is also valid as the composition operator is transparent w.r.t. validity.[6]

The next step is to show that the set of valid instantiations for Q_2 (over $\mu(D_1)$) and for Q_2^t (over D_1) are identical. First, assume Q_2 is covered by the head of one rule in μ. Given a valid instantiation ι' for Q_2, let us consider its projection over the leaf variables of $\mu(D_1)$, let us call it $\Pi(\iota')$. We can define an isomorphism τ from nodes of this projection to the nodes in $\Pi(\iota)$. It is easy to check that this isomorphism ensures that the set of valid instantiations for Q_2 and Q_2^t are identical.

Then we consider Q_2 spanning multiple rule heads in μ. Here a valid instantiation ι over $\mu(D_1)$, is the join of the single instantiations against multiple rule heads. The joins obviously preserve the predicates and ensures that the constructing procedure for ι' is again valid.

We now show that the claim holds for the forward direction. Let μ, Δ_1, Δ_2, Q_1, Q_1^t be as defined above. Then, for every $D_2 \in inst(\Delta_2)$: $Q_1^t(D_2) = \bigcap_{D_1^k : \mu(D_1^k) = D_2} Q_1(D_1^k))$.

Forward translation (Fig. 16) is analogous to computing the certain answers to a query Q_1. Let ι: $\mathcal{V} \mapsto \mathcal{D}$ be a variable instantiation as a function that maps each variable $\in \mathcal{V}$ in the query (whether it be Q_1 or Q_1^t) to a value in a database D. Intuitively, ι must be valid for any database D that is consistent with D_2. To represent all possible databases consistent with D_2, we must define all possible pre-images D_1^k such that $\mu(D_1^k) = D_2$. Since there may be several pre-images, for each pre-image D_1^k, ι^k is a valid instantiation from Q_1 to D_1^k. Similarly, ι' is a valid instantiation from Q_1^t to D_2. Thus, for k instances of Δ_1, we must consider the intersection $\iota = \bigcap_{D_1^k} \iota^k$, the intersection of valid instantiations still being a valid instantiation. We can define an isomorphism τ from nodes of ι to the nodes in ι'. It is easy to check that this isomorphism ensures that the set of valid instantiations for Q_1 and Q_1^t are identical.

The above held for a query Q_1 spanning a single rule body. With a query Q_1 spanning multiple rule bodies, the same consideration made for the backward translation holds. □

6 Experimental study

This section studies the performance of HePToX's mapping generation, query translation, and query processing. We also show HePToX's scalability w.r.t. the number of peers and the data heterogeneity of the network. All experiments adhere to

[6] Indeed, μ^{-1} can be seen as the translation of a query covering the entire rule head against the rule body, whereas ι is a query against the rule body. Thus, the composition operator would be exactly the *Cartesian product* of these two queries.

Algorithm TransForward.
Input: a join of tree pattern queries Q_i, mapping rule set from $\Delta_1 \rightarrow \Delta_2$
Output: one translated query in the form of join of TPs.
1. For each TP Q_i, translate it: let $Q'_i = \mathbf{TransTPForward}(Q_i)$.
2. Rename the variables apart in the translated TPs.
3. Add join conditions to the set of translated TPs (with variable renaming).
4. Chase the set of TPs along with schema key constraints, the join conditions and other query conditions, merging nodes as necessary.
5. Obtain contractions of previous query Q' by recursively removing appropriate dummy nodes bottom up.
Let Q^t be the resulting query obtained by replacing Skolem terms with associated node tags.
6. Translate the final merged TP (or set of TPs) Q^t into XQuery.

Procedure TransTPForward.
1. For each mapping rule $\mu_j \equiv rh_j \leftarrow rb_j$
a. **Expansion:** Find a substitution and expand Q_i to Q^e_{ij} by matching the rule body rb_j to the query pattern Q_i
For each node in Q^e_{ij} and not in Q_i, mark it as dummy
If it is a distinguished node in Q_i, mark it as distinguished in the expanded query Q^e_{ij}
b. **Translation:** Translate each Q^e_{ij} into Q^{t*}_{ij} by applying μ_j; mark dummy and distinguished nodes accordingly
2. Stitch together all Q^{t*}_{ij} to obtain resulting query Q^t_i. Nodes with same or unifiable Skolem terms get stitched by looking at the corresponding substitutions.
Procedure TransTPBackward.
1'.For each mapping rule $\mu_j \equiv rh_j \leftarrow rb_j$
a'.Find all substitutions and expand Q_i to Q^e_{ij} by matching the rule head rh_j to the query pattern Q_i and by looking up the correspondence in the rule body rb_j
For each node in the rule head rh_j and not in query pattern Q_i, mark it as dummy
If it is a distinguished node in Q_i mark it as distinguished in the expanded query Q^e_{ij}
b'.Translate each Q^e_{ij} into Q^{t*}_{ij} by projecting every variable of rule head rh_j to rule body rb_j; mark dummy and distinguished node accordingly
2'. Stitch together all Q^{t*}_{ij} to obtain resulting query Q^t_i. Nodes with same variables get stitched by looking at the corresponding substitutions.

Fig. 16 Query translation algorithm

a common scenario, where each peer maps to a set of acquaintances, using the mapping rules in Sect. 4.

6.1 Experimental guidelines

In all the experiments, we consider a P2P network consisting of several peers. The mappings in HePToX connect a peer to a set of acquaintances, thus forming a mapping graph. It is not necessary to build a strongly connected graph, as it is sufficient that a transitive connection exists between every pair of nodes. Mappings between two sources can then be obtained by composing the mappings between all the peers that connect the two sources. There may exist alternative mapping paths between the two peers. Because of that, the network has a considerable resilience to node failures, since a failing or leaving node on a mapping path does not invalidate the remaining mapping paths.

Currently the *shortest path* is chosen during query evaluation. Other criteria may be used, such as the size of intermediate answers or the coverage of the query by the schema mappings; this is beyond the scope of this paper. HePToX marks each query with a global unique id; peers will not process queries with previously seen ids, thus handling cycles that occur in a semantic path.

The network gracefully handles insertions of new nodes. Each peer joining the network maps its local schema to that of a few acquaintances. Any existing peer in the network may similarly map its schema to this new peer's schema. Recall that, although mappings are unidirectional, query processing can take place along and against the mappings.

Thanks to P2P load balancing, HePToX's peers execute queries on their local data and retrieve the query results that are relatively concise, thus reducing the network load. In this section, we demonstrate that HePToX query evaluation is efficient and local resources consumption is minimized as each peer only needs to remember the addresses and mappings of its acquaintances. This localization means that a user asking a query will not need to know where the answers are exactly coming from, nor does the user know how many peers respond to his query, thus making the P2P paradigm a natural setting for query reformulation in HePToX.

6.2 Implementation and setup

Emulab [15], a network emulation testbed, to get a realistic P2P network. Emulab consists of a collection of PCs, the network delay and bandwidth of which can be set at will. Emulab allows the full allocation of a real machine's resources. We observed that XML query answering on each peer is computationally demanding, and cannot just rely on partially allocated machines, as in other emulators, e.g., Planetlab, justifying our choice of Emulab. We could get `50` real machines in total from the Emulab network. We chose a 70 ms delay and a 50 MB bandwidth to simulate as much as possible the real networks behavior. We mounted FreePastry [36] as the network protocol. Compared to unstructured P2P networks (e.g., gnutella), FreePastry offers scalable and efficient P2P routing. Its O(log N) routing complexity, and O(log N) routing table size is at least as efficient as CAN, Tapestry, Chord etc. The current implementation of HePToX exhibits high modularity; it can run with any XML Query engine

Table 2 Query #, query description, and query mapping coverage (% of QMC)

Query#	Query description	QMC (%)
Q_1	Selection with 1 filter, Mich. QR3, etc.	11.4
Q_2	Selection with 2 filters, Mich. QS5, etc.	13.7
Q_3	Selection with 3 filters, Mich. QS16, etc.	20.7
Q_4	Selection with 2 filters (1 nested), Mich. QS18, etc.	11.5
Q_5	Selection with 3 filters (2 nested), Mich. QS34 etc.	11.8
Q_6	1 Join, Selection with 2 filters, Mich. QJ1, etc.	28.6
Q_7	1 Joins, Selection with 1 filter, Mich. QJ3, etc.	28
Q_8	3 Joins, Selection with 1 filter, No corresp. Mich.	62.5
Q_9	6 Joins, Selection with 1 filter, No corresp. Mich.	100
Q_{10}	9 Joins, Selection with 1 filter, No corresp. Mich.	100

by implementing a simple API. We chose QIZX [40] as an XML query engine, as this is the fastest open-source XQuery engine we could find. We used FreePastry vs.1.3.2 and QIZX vs.0.4p1, respectively. HePToX is written in JAVA, thus making it cross-platform.

6.3 Guiding principles

The experiments are conducted according to the following guidelines: (i) each peer joining the network is equipped with the peer schema and the peer data adhering to that schema; (ii) each peer chooses other peers as its acquaintances, and these acquaintances in turn see it as their acquaintance; (iii) a peer evaluating a query translates the query to the schema of each acquaintance, according to our algorithm, and ships the translated query to that acquaintance; (iv) forwarding of queries by a peer stops as soon as the peer realizes it already processed an incoming query request, to avoid cycles. We broadcast a query to all the other acquaintances, as in real P2P scenarios. More optimized broadcasting can be applied, and is beyond the scope of this work.

Datasets and queries used for experiments. To probe the efficiency of query translation (Fig. 16), we considered both synthetic and real XML datasets. As a synthetic dataset, we derived 9 *restructured variations* of XMark [41], covering a variety of structural transformations. This produced 10 different XMark schemas randomly scattered across the network. Detailed description of the schemas can be found in [23]. We modified the XMark xmlgen code accordingly to generate the datasets conforming to the schemas above with an average size of 50 MB. We ran a comprehensive set of 10 queries on the XMark schemas: 7 queries are from the UMichigan XML benchmark queries (adapted to XMark schemas) and 3 queries are multiple joins queries of increasing complexity. A summary description of these queries is in Table 1; complete query specifications are in [23]. The UMichigan XML benchmark queries were more suitable to probe the effectiveness of our algorithms than the XMark queries themselves, since they build around the XML data structures and let us leverage the heterogeneity of our schemas variations. In order to study the performance of the algorithms on a collection of queries against a common data set, we adapted the UMichigan XML benchmark queries to the XMark dataset and use the latter data set for all our queries.

Similarly, to see the effectiveness of our translation algorithm on real data we used the DBResearch collection of 19 XML schemas used in Piazza [44]. For this dataset, we ran the same queries used in Piazza experiments, which amount to multiple-joins queries of increasing complexity. The description of these queries can be found at [23].

We measured the average number of rules across any pair of schemas; it is 8.67 for the synthetic dataset. Table 1 outlines the % of *query mapping coverage* (QMC) for each query as the percentage of rules traversed by that query divided by the average number of rules on any schema pair. Table 1 shows that all the benchmark queries QMC is up to 25% of the total coverage, whereas ad-hoc join queries are close to 100% of the total coverage. Thus, the queries in Table 1 cover all cases and satisfactorily span the QMC %.

6.4 Rule inference algorithm in HePToX

HePToX's GUI [11] allows the user to draw a few arrows/boxes, and generate the corresponding rules accordingly. To test the performance of the algorithms in Sect. 4, we measured the time to infer the rules for XMark schemas variations (DBResearch schemas, respectively), having an average of 65 (7) arrows between them. Our algorithm generates an average of 9 (3) rules in about 45 (25) ms (tested on a P4 machine, with 3.0 GHz and 2 GB memory). We could not compare the scalability of our rule inference algorithm with the corresponding mapping generation algorithm in Clio [24], as the latter is under copyright. We instead compare the

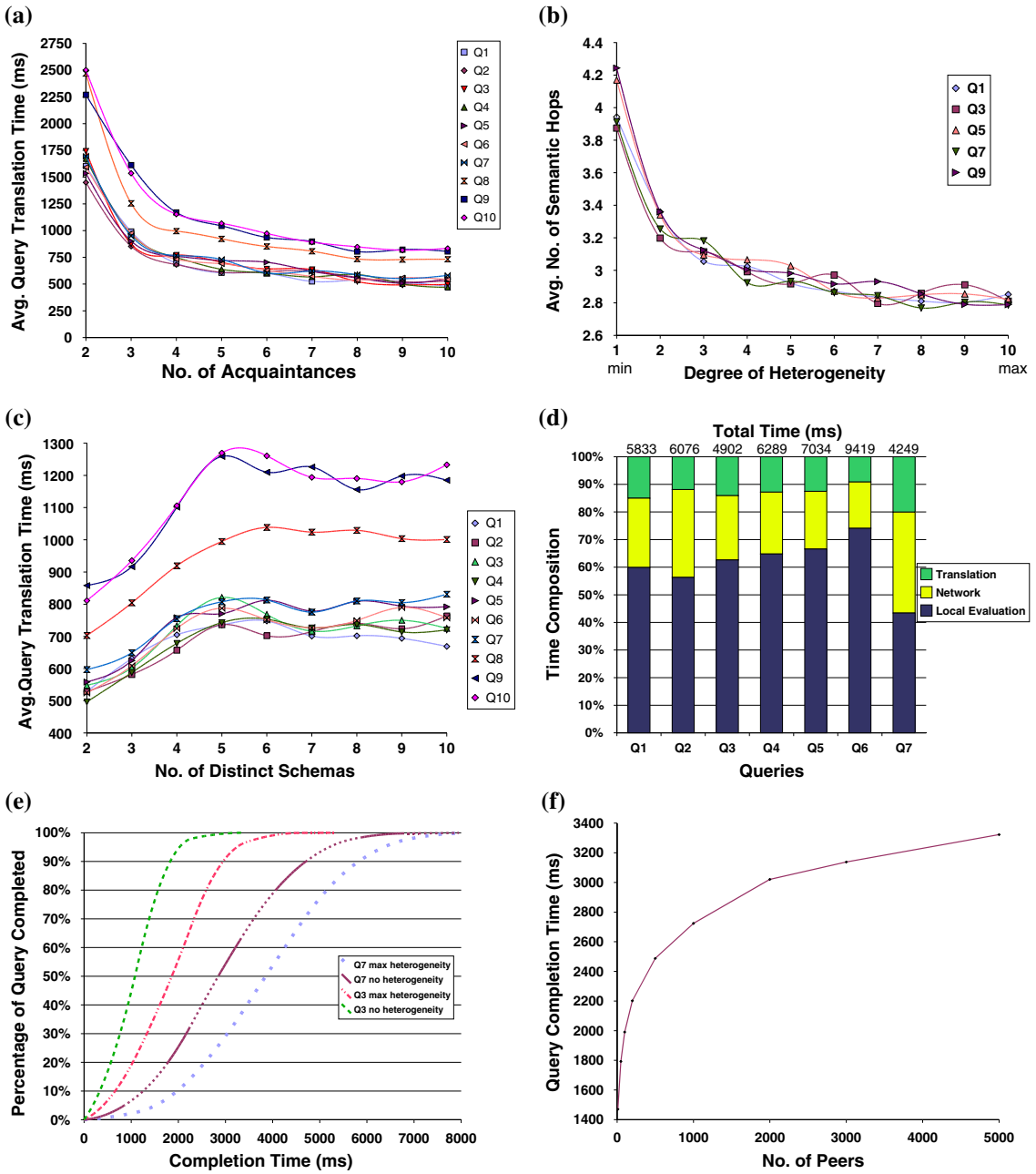


Fig. 17 HePToX Query translation and query performance for XMark dataset: **a** Average QT time w.r.t. average number of acquaintances. **b** Average number of semantic hops w.r.t. degree of homogeneity. **c** Average QT time w.r.t. total number of distinct schemas. **d** Time composition for querying. **e** Query completion w.r.t. timeout. **f** Scalability of the query completion time w.r.t. number of peers

usability of the two systems; we describe our experience in Sect. 6.8.

6.5 Query translation in HePToX

In this experiment, we probe the effectiveness of the query translation algorithm under different network configurations. In particular, we realized different parameters may affect query translation time: (a) the average number of acquaintances across all sets; (b) the degree of 'heterogeneity' (ranging from all acquaintances having the same schema to all acquaintances having different schema); and (c) the overall number of distinct heterogeneous schemas scattered across the network. In this experiment, we investigate how (a)–(c) impact query translation time.

In experiment (a), we consider there to be a total of distinct 10 schemas in the network, and measure the query translation (QT) time when the average number of acquaintances on

each peer varies from 2 to 10. The result, reported in Fig. 17a, showed us that in order to keep the number of translations reasonably low, we have to choose an average number of acquaintances of (at least) 4. This number of acquaintances for a 50 peers network is indeed sufficient to get an acceptable average length of semantic paths for that network, as also confirmed by next experiment (b). Indeed, Fig. 17b plots the average number of hops (or length of semantic paths) achieved when the average number of acquaintances is equal to 4 (few queries are reported to avoid clutter). The x axis in such a case represents the degree of 'heterogeneity' increasing from 1 to 10, meaning that 1 to 10 acquaintances have a distinct schema. It can be noted that the average length of semantic paths (reaching 3) stabilizes with (at least) a degree of heterogeneity of 3. These two experiments let us choose an average number of acquaintances equal to 4 in the remainder.

Figure 17c shows how the total number of distinct schemas present in the network affects the average QT time. This experiment shows that the average QT time grows almost linearly with the number of distinct schemas up to 5 and stabilizes after 5. It lets us conclude that variations of the QT time can be appreciated for a number of schemas less or equal to the number of acquaintances (i.e., 4) and are blurred otherwise.

6.6 Query performance in HePToX

The next experiment examines the minimal overhead introduced by our translation algorithm all along the query answering process. In Fig. 17d, we highlight the various time components (%) taken by query translation, network delay, and local query answering, respectively, while the actual times (in *ms*) are reported on top of each bar. It can be noted that query translation takes a negligible time if compared to network delay and local query answering. The latter, essentially due to QIZX, was the bottleneck for all queries and caused the crashing of the most complex ones. Query answers to Q_8, Q_9, and Q_{10} are omitted in this plot, since they neither completed nor yielded any answers within the given timeout. We tried different query engines before choosing QIZX, which is considered the fastest one, thus this behavior was definitely outside our control.

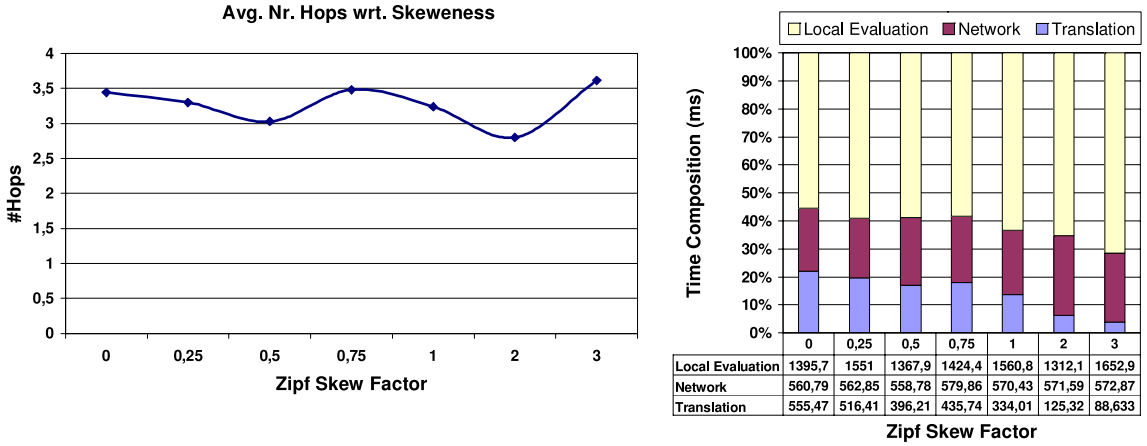
The next experiment complements the previous one by measuring the % of query completed within a specified timeout with no heterogeneity/the maximum heterogeneity in the network, respectively. Figure 17e shows the % of query completed when the peers have all different schemas (maximum heterogeneity) w.r.t. the baseline case when all peers exhibit the same schema (no heterogeneity). We can see that for instance the difference between the two curves for queries Q_3 and Q_7 is at most 1,000 ms, showing the overall negligible impact of translation over query processing. In the above experiments, we have considered a uniform schema distribution, in which each peer owns the same number of schemas. We tested our system with a skewed schema distribution in which the participating peers have an unbalanced number of schemas. We used the Zipfian distribution to simulate the skew, as the latter closely simulates the real cases. Figure 18(left) shows the average number of hops for different skew factors (from 0 (uniform distribution) to 3). We can notice that this number decreases to a steady state, thus showing that more local evaluation (and less translations, resp.) takes place as we increase the skewness. This is confirmed by the time composition shown in Fig. 18(right) for such non-uniform distribution. This experiment was done by executing Q1 on 50 Emulab peers.

6.7 Scalability and network churn in HePToX

The next experiment in Fig. 17f shows the scalability of HePToX P2P databases. We report the query completion time (including the query translation time, query evaluation and network delay) when varying the number of peers in the network up to 5,000 peers. This experiment was executed on the Pastry simulator alone, as Emulab real machines would not be enough. It can be observed that query completion follows a quite regular logarithmic curve.

To study the effect of joining/leaving peers on HePToX query translation and query completion time, we have run another experiment in Pastry by considering an initial number of peers equal to 1,000. In such experiment, as peers are leaving, they also take away the acquaintance lists of their former acquaintances. The peer's number of acquaintances never drops below 5, and, in case this happens, the peer creates new acquaintances. We can observe in Fig. 19 the time composition by varying the number of leaving peers from 0% to 50% (i.e., until 500 peers are left). Times gracefully decrease while increasing the percentage of leaving peers in the network. We repeated the same experiment on the result network of 500 peers by varying the number of joining peers from 0% to 50% (i.e., until reaching a number of 1,000 peers) and we observed a symmetric trend. Due to the lack of space, this result is omitted.

We next considered a *real dataset, DB research*. We ran the previous experiments on all Piazza queries; the results were very similar to the previous results, so we only report here the time composition graph in Fig. 20. Modulo the fact that HePToX and Piazza have a different syntax for their mapping languages, we obtained the same 29 mappings employed in Piazza. It can be noted that the behavior of our query translation algorithm is similar for both synthetic and real data. The times reported in Fig. 20 are reduced compared with the XMark datasets due to the smaller sizes of DBResearch data (up to only 27 KB per peer).



	0	0,25	0,5	0,75	1	2	3
Local Evaluation	1395,7	1551	1367,9	1424,4	1560,8	1312,1	1652,9
Network	560,79	562,85	558,78	579,86	570,43	571,59	572,87
Translation	555,47	516,41	396,21	435,74	334,01	125,32	88,633

Fig. 18 Average number of hops and time composition in a skewed P2P schema distribution

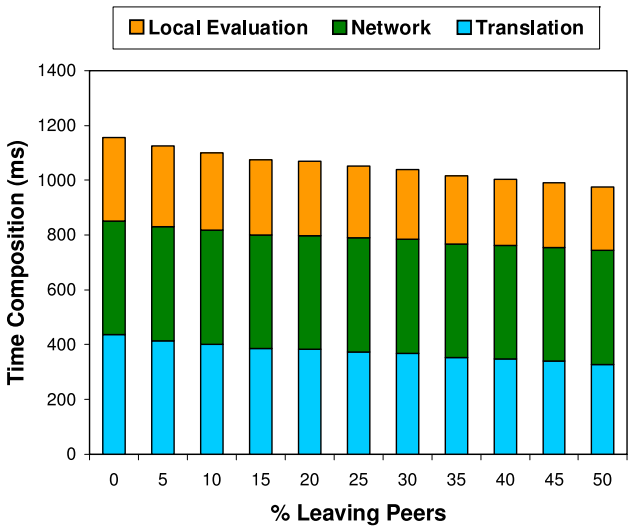


Fig. 19 Time composition in case of network churn in HePToX with XMark dataset

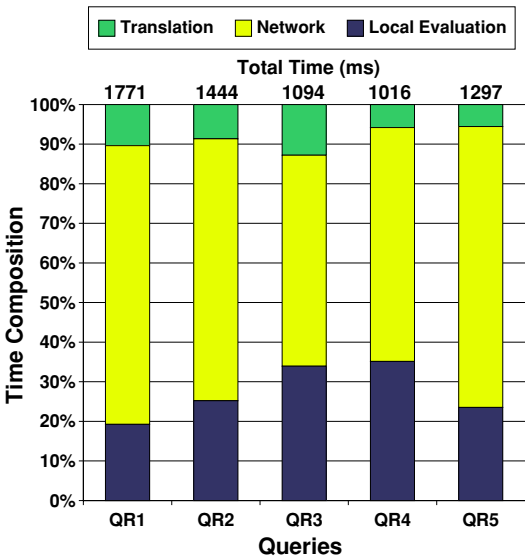


Fig. 20 Time composition for querying in HePToX for the DBResearch data set

6.8 Usability

One final issue to be explored is how usable is HePToX–does HePToX's system of boxes and arrows reduce the user's effort in creating a mapping? To do so, we used the 11 usability scenario from STBenchmark [1]. In particular, we used HePToX's visual interface to implement the STBenchmark basic mapping scenarios. We then compared these results to the results on the same scenarios as conducted on Clio in [1]. Although the use of the resultant mappings are different—Clio concentrates on translating data, whereas HePToX translates queries—the amount of effort to create the initial mappings is worth comparing.

STBenchmark provides a simple usability model, SU, which we adopt. The SU model quantifies human effort as a quadruple (L,S,D,K), where L is the number of dragging actions, S and D are the number of single mouse clicks, and double mouse clicks, respectively, and K is the number of keystrokes used for text input [1]. For example, the tuple (1,3,5,11) says that it required 1 dragging action, 3 single mouse clicks, 5 double mouse clicks, and 11 keystrokes to implement a mapping scenario.

Given a quadruple (L,S,D,K), the SU model associates an effort cost using the equation $cost = (4L + S + 2D + 4K)$ [1]. The dragging and keystrokes are assigned higher costs based on the findings in [1] that it is easier to make mistakes in dragging and keystrokes than it is to click the mouse. The double click is assigned twice the cost of the single click since it requires twice the effort.

For this experiment, we downloaded the 11 basic mapping scenarios from the STBenchmark website [42]. For each mapping scenario, STBenchmark provided a diagram of an expected mapping and transformation rule along with source

Table 3 Testing the usability of HePToX and Clio

Scenario/mapping system	HePToX		Clio	
	Effort	Cost	Effort	Cost
Copy	(3,3,0,0)	15	(0,4,0,0)	4
Constant value generation	+	+	(0,6,0,17)	74
Horizontal partition	(4,4,0,0)	20	(0,22,2,21)	110
Surrogate key assignment	(4,4,0,0)	20	(0,42,2,39)	202
Vertical partition	+	+	(0,7,0,0)	7
Unnesting	(4,4,0,0)	20	(0,7,0,0)	7
Nesting	(4,4,0,0)	20	(0,18,2,0	22
Self Joins	*	*	*	*
Denormalization	+	+	(0,23,2,1)	31
Keys and object fusion	(9,9,0,0)	36	(0,30,4,0)	38
Atomic value changes	(6,6,0,0)	30	(0,20,0,45)	200

A "+" indicates that the mapping was able to be created outside of the user interface but that a small extension to the interface would be necessary to create the mapping inside the interface. A "*" indicates that a substantial extension would be required to create the mapping inside the interface

and target descriptions in XSD. Since HePToX used DTDs, we used Altova XMLSpy 2009 [2] to convert from XSD to DTD, followed by manual corrections.

We then had an experimenter practice each mapping scenario to become familiar with HePToX's user interface and create the mapping with the least cost. This result was then compared with results of Clio from [1], which were performed under similar conditions. The results are shown in Table 3.

We can observe that, although Clio was able to implement more scenarios, HePToX required significantly less effort on the majority of the scenarios that it could implement. The scenarios in which Clio has a lower cost were the 'Copy' and 'Unnesting' scenarios, in which Clio utilizes the automatic generation of correspondences, which can be verified by the administrator. Thus, the suggested matching generated by Clio was only needed to be confirmed with single mouse clicks on these special cases. In the other cases, where Clio was unable to use its automatic matching module, we observed that the key difference was that HePToX is basically click-and-drag oriented and Clio is instead click-and-select oriented. Based on these experiments, we see that click-and-drag requires less effort.

Although it is algorithmically capable of handling such mappings, the HePToX interface was unable to implement the 'Constant value generation', 'Vertical partition', and 'denormalization' scenarios because they required heavy modifications beyond the capabilities of the simple prototype user interface. As an example, HePToX does not provide a graphical user interface to relate the two vertical partitions of a target schema by the referential constraint on one of the attributes. The modifications required to do so, and are marked with a "+" in Table 3. Both systems were not able to implement the 'Self join' scenario because the latter requires the system to duplicate the schema through the visual interface. Since both systems' interfaces can only load single source and target schemas, changing this would require a substantial change to their user interfaces.

As a first step toward a usability model for constructing mappings, the SU model makes one big assumption: it does not take into account human errors and 'thinking time' during mapping design. Thus, the SU model has to assume that the mapping designer is an expert who is familiar with the visual interface and makes no mistakes [1]. For such a reason, future benchmarks accounting for such extensions would be urgently needed.

7 Related Work

Data exchange and P2P data integration systems. The *Clio* data exchange project is one of the pioneering schema mapping tools [32,37]). They were the earliest to propose visual schema correspondences and developed an efficient algorithm for discovering the source-to-target tuple-generating dependencies (or tgds) from the lines. Their framework captures both relational and nested relational schemas.

Clio derives a set of tgds from a set of lines between a source schema and a target schema. Applying these transformation computes a target instance that adheres to the target schema and to the correspondences. Similarly, HePToX derives a set of TreeLog mapping rules from element correspondences (i.e., boxes and arrows) between two schemas. TreeLog rules are similar in spirit to tgds, although TreeLog has a higher order syntax that facilitates mappings between data and schema elements. However, the problems solved by

Clio and HePToX are different. In a PDMS, the goal is not data exchange (i.e., translating data), but gathering answers to queries from any peer (i.e., translating queries). In principle, one could translate the data from every peer to the querying peer and then answer the query there. But this is impractical, as it would require translating every peer's data to every other peer's schema.

Clio has been extended [46] to handle 'XML target query answering', i.e., an XML query posed against the target schema must be translated into a set of queries against the sources, via the mappings from source to target and a set of target constraints. To the best of our knowledge, they do not handle data-metadata mappings in query translation. Moreover, their query rewriting algorithm is against the direction of the mappings, and thus equivalent to the backward direction in HePToX, disregarding target constraints. The forward direction HePToX query rewriting algorithm is entirely new, and not addressed in any works on XML query rewriting [20,46].

Earlier versions of Clio [37] use the partitioned normal form (PNF) as the default grouping mechanism. PNF groups nested set of elements by the atomic elements in the upper levels. Consider an example adapted from the schemas in [37] with source *src*: Emps → Emp*, Emp → A B C, and target *tgt*: Emps → Emp*, Spouses → Spouse*, Emp → A B* E, Spouse → E C, where there is a keyref/key constraint from Emp/E to Spouse/E. Applying PNF in Clio, creates the value of E in schema *tgt* as a function of the values of A and C, whereas in HePToX creates a function of A. There are natural examples justifying both choices. For example, let A = empName and C = spouseName. If B = child, creating E as a function of A and C nicely groups children of an employee and spouse. But, if B = employee's hobby, which is unrelated to spouse, then creating E as a function of A is more meaningful.

A recent Clio extension to nested mappings [19] allows grouping to be further customized and declaratively changed. This allows users to modify the default PNF grouping condition, and is especially useful in schema evolution applications, when two similar source schemas are mapped to the same target schema. Since our aim in HePToX is query reformulation rather than data exchange, we did not add different grouping conditions to our TreeLog rule language.

The latest version of Clio addresses data exchange with data-metadata correspondences [24], where NDOS (nested schemas with dynamic elements) define data exchange between a source database and a target database. As we already discussed, HePToX focuses on query reformulation and answering in PDMS rather than on data exchanged. We devised TreeLog, an extension of SchemaSQL/SchemaLog [4], that has, among the other features, the ability to specify views whose schema is dynamic, in the sense that it exploited in the target schema data instances that are part of the source database. Specifically, since we handle query translation in a P2P setting, in which the peers come with their own schema, the presence of dynamic elements in the target is not meaningful (because we never need to create a target schema dynamically on a peer). Therefore, although HePToX (via TreeLog) has the expressive power to define mappings where the target schema is dynamic, in a PDMS knowledge of the instances cannot be exploited while creating such correspondences, while it has to be in a mapping tool, such as Clio [24].

Finally, a number of commercial visual programming tools help the user to produce mappings, such as Altova MapForce, BEA WebLogic, Altova Stylus Studio, IBM WebSphere, and Microsoft BizTalk Mapper. Such tools require substantial user intervention and have been compared in [1].

In *Piazza* [20,21], each peer stores semantic mappings and storage descriptions. Semantic mappings are equalities or subsumptions between query expressions, provided in XQuery. Storage descriptions are equalities or subsumptions between a query and one or more relations stored on a peer. In HePToX, the exact mapping rules are derived automatically from correspondences, which are very intuitive for the user. Piazza's query reformulation is quite different from HePToX's. In Piazza, semantic mappings are first used to do query rewriting using the MiniCon algorithm [39]. When semantic mappings cannot be applied further, storage descriptions are used to do query reformulation. The result of this phase is a reformulation of peer relations into stored relations, which can be either in GAV or LAV style. Query routing in Piazza requires centralized index that scales with the number of attributes of individual peers. In contrast to Piazza, HePToX is totally decentralized and its scalability is less than linear (i.e., logarithmic, as in DHT-based systems).

The semantics of HePToX's forward query translation is similar to answering queries using views [29]. However, we can leverage Skolem functions and the form of the mapping rules to perform forward translation efficiently. Benedikt et al. [7] and Bohannon et al. [10] also have studied mappings between schemas, including recursive ones and node-to-path mappings. None of the above works, including Piazza, handles schema mappings involving data-metadata conflicts.

Orchestra [26] extends PDMSs for life scientists. It focuses on provenance, trust, and updates. While it can be extended to XML, it uses the relational model. Orchestra's mapping rules translate from tgds to Datalog, rather than HePToX's mapping rules which translate from a visual language to TreeLog. Unlike HePToX, which supports the user in easily creating the mapping between schemas, Orchestra relies on other systems to create the initial mappings. Moreover, the Q system, which is the query module in Orchestra, focuses on keyword queries rather than on XQuery queries.

Calvanese et al. [12] address data interoperability in P2P systems using expressive schema mappings, also following the GAV/LAV paradigm, and show that the problem is in PTIME only when mapping rules are expressed in epistemic logic. [14] studies finding a minimal query reformulation between relational and XML schemas. Consistency of XML data exchange for tree patterns is studied in [6].

Query answering relies on the notion of certain answers, and leaf nodes bound in the mappings are captured into functional dependencies. The tractable cases for checking mapping consistency are identified whenever mapping expressions do not include the descendant axis and wildcard.

Fagin et al. [18] show that composition of finite sets of source-to-target tuple-generating dependencies (tgds) is always definable by a second-order tgd. Mappings between data and schema items as in HePToX are in terms of TreeLog rules, and composition is not considered. We are able to directly generate mappings between data and schema items across schemas, from given correspondences. Our mapping language, like SchemaLog [4], is semantically reducible to first order. A recent paper [19] also considers nested mappings, which are not handled here. Finally, the primitive groups used in mapping rule inference (Sect. 4) are similar to the primary paths used in Clio [37].

Schema-matching systems. Automated techniques for schema matching (e.g., CUPID [16,30,38]) can output elementary schema-level associations by exploiting linguistic features, context-dependent type matching, similarity functions etc. These associations could constitute the input of our rule inference algorithm if the user does not provide the arrows. Bohannon et al. [9] discusses how to infer conditions for contextual schema matching. Such conditions could be employed as input to the data-metadata correspondences used in HePToX. Again, we assume that the above correspondences are provided by a user rather than output by a tool, such as the one in [9].

Ontology-matching systems. Ontology alignment also creates a mapping, but it is very different from the mappings created in HePToX. Most previous work focuses on finding simple equivalences. We present some typical work ontology alignment, and refer the reader to [27] for a survey. *OntoMorph* [13] translates symbolic knowledge between different knowledge representations through user-driven transformation rules. *Prompt* [34] finds corresponding concepts by refining an initial mapping (pairs of anchors) given by users or some simple linguistic matching approaches. The philosophy followed by *Prompt* is similar to that of *Similarity Flooding* [31]. *FCA-Merge* [43] is an alignment technique that depends on external resources to find Is–A relationships between concepts. However, since the formal context is built upon the generalization/specialization hierarchy of the concepts, this approach could not be extended to the more complex mappings that HePToX creates.

8 Conclusions and future work

We have presented the HePToX P2P XML database system, focusing on the following key conceptual contributions: (i) an algorithm for inferring mapping rules, expressed in a higher-order logic TreeLog, from correspondences between heterogeneous peer schemas, specified via boxes and arrows, covering data-metadata interplay between schemas; (ii) a precise and intuitive semantics for query evaluation in a P2P setting; (iii) a query translation algorithm that is correct w.r.t. this semantics and is efficient, as revealed by the detailed experimentation. It is interesting to investigate extension of the techniques developed here to handle larger classes of mappings and queries.

Acknowledgments Many thanks to Alon Halevy and Igor Tatarinov for sharing their DBResearch data sets with us, and to Renée Miller for invaluable comments.

References

1. Alexe, B., Tan, W.C., Velegrakis, Y.: Stbenchmark: towards a benchmark for mapping systems. PVLDB **1**(1), 230–244 (2008)
2. Altova XMLSpy: http://www.altova.com (2009)
3. Amer-Yahia, S., Cho, S., Lakshmanan, L., Srivastava, D.: Minimization of tree pattern queries. In: SIGMOD, pp. 497–508 (2001)
4. Andrews, A.J., Lakshmanan, L.V.S., Shiri, N., Subramanian, I.N.: On implementing schemaLog—a database programming language. In: CIKM, pp. 309–316 (1996)
5. Arenas, M., Kantere, V., Kementsietsidis, A., Kiringa, I., Miller, R., Mylopoulos, J.: The hyperion project: from data integration to data coordination. SIGMOD Rec. **32**(3), 53–58 (2003)
6. Arenas, M., Libkin, L.: XML data exchange: consistency and query answering. In: PODS, pp. 13–24 (2005)
7. Benedikt, M., Chan, C., Fan, W., Freire, J., Rastogi, R.: Capturing both types and constraints in data integration. In: SIGMOD, pp. 277–288 (2003)
8. Bernstein, P.A., Giunchiglia, F., Kementsietsidis, A., Mylopoulos, J., Serafini, L., Zaihrayeu, I.: Data management for peer-to-peer computing: a vision. In: WebDB, pp. 89–94 (2002)
9. Bohannon, P., Elnahrawy, E., Fan, W., Flaster, M.: Putting context into schema matching. In: VLDB, pp. 307–318 (2006)
10. Bohannon, P., Fan, W., Flaster, M., Narayan, P.: Information preserving XML schema embedding. In: VLDB, pp. 85–96 (2005)
11. Bonifati, A., Chang, E., Ho, T., Lakshmanan, L.V.S., Pottinger, R.: HEPTOX: marrying XML and heterogeneity in your P2P databases. In: VLDB, pp. 1267–1270 (2005)
12. Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: Logical foundations of peer-to-peer data integration. In: PODS, pp. 241–251 (2004)
13. Chalupsky, H.: Ontomorph: a translation system for symbolic knowledge. In: KR, pp. 471–482 (2000)
14. Deutsch, A., Tannen, V.: Reformulation of XML queries and constraints. In: ICDT, pp. 225–241 (2003)
15. Emulab. http://www.emulab.net
16. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB J. **10**(4), 334–350 (2001)
17. Fagin, R.: Inverting schema mappings. In: PODS, pp. 50–59 (2006)

18. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.C.: Composing schema mappings: second-order dependencies to the rescue. In: PODS, pp. 83–94 (2004)
19. Fuxman, A., Hernández, M.A., Howard, C.T., Miller, R.J., Papotti, P., Popa, L.: Nested mappings: schema mapping reloaded. In: VLDB (2006)
20. Halevy, A.Y., Ives, Z.G., Suciu, D., Tatarinov, I.: Schema mediation in peer data management systems. In: ICDE, pp. 505–516 (2003)
21. Halevy, A.Y., Ives, Z.G., Mork, P., Tatarinov, I.: Piazza: data management infrastructure for semantic web applications. In: WWW, pp. 556–567 (2003)
22. HepApp: http://staff.icar.cnr.it/angela/VLDBJappendix.pdf
23. HepTox: http://www.cs.ubc.ca/labs/db/heptox/exp.htm
24. Hernández, M.A., Papotti, P., Tan, W.C.: Data exchange with data–metadata translations. PVLDB **1**(1), 260–273 (2008)
25. Hull, R., Yoshikawa, M.: ILOG: declarative creation and manipulation of object identifiers. In: VLDB, pp. 455–468 (1990)
26. Ives, Z.G., Green, T.J., Karvounarakis, G., Taylor, N.E., Tannen, V., Talukdar, P.P., Jacob, M., Pereira, F.: The orchestra collaborative data sharing system. SIGMOD Rec **37**(3), 26–32 (2008)
27. Kalfoglou, Y., Schorlemmer, M.: Ontology mapping: the state of the art. Knowl Eng Rev **18**(1), 1–31 (2003)
28. Kementsietsidis, A., Arenas, M., Miller, R.: Mapping data in peer-to-peer systems: semantics and algorithmic issues. In: SIGMOD, pp. 325–336 (2003)
29. Levy, A.Y., Mendelzon, A., Sagiv, Y., Srivastava, D.: Answering queries using views. In: PODS, pp. 95–104 (1995)
30. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with cupid. In: VLDB, pp. 49–58 (2001)
31. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: ICDE, pp. 117–128 (2002)
32. Miller, R.J., Haas, L.M., Hernández, M.A.: Schema mapping as query discovery. In: VLDB, pp. 77–88 (2000)
33. Ng, W.S., Ooi, B., Tan, K., Zhou, A.: PeerDB: a P2P-based System for distributed data sharing. In: ICDE, pp. 633–644 (2003)
34. Noy, N., Musen, M.: Prompt: Algorithm and tool for automated ontology merging and alignment. In: AAAI, pp. 450–455 (2000)
35. Papakonstantinou, Y., Abiteboul, S., Garcia-Molina, H.: Object fusion in mediator systems. In: VLDB, pp. 413–424 (1996)
36. Pastry: http://research.microsoft.com/~antr/Pastry/
37. Popa, L., Velegrakis, Y., Miller, R.J., Hernández, M.A., Fagin, R.: Translating web data. In: VLDB, pp. 598–609 (2002)
38. Pottinger, R., Bernstein, P.A.: Merging models based on given correspondences. In: VLDB, pp. 826–873 (2003)
39. Pottinger, R., Halevy, A.: MiniCon: a scalable algorithm for answering queries using views. VLDB J. **10**(2–3), 182–198 (2001)
40. Qizx: http://www.xfra.net/qizxopen/
41. Schmidt, A., Waas, F., Kersten, M., Carey, M., Manolescu, I., Busse, R.: XMark: a benchmark for XML data management. In: VLDB, pp. 974–985 (2002)
42. STBenchmark: http://www.stbenchmark.org
43. Stumme, G., Maedche, A.: FCA-MERGE: bottom-up merging of ontologies. In: IJCAI, pp. 225–230 (2001)
44. Tatarinov, I., Halevy, A.: Efficient query reformulation in peer-data management systems. In: SIGMOD, pp. 539–550 (2004)
45. Ullman, J.: Principles of Database and Knowledge-Base Systems. Computer Science Press (1988)
46. Yu, C., Popa, L.: Constraint-based XML query rewriting for data integration. In: SIGMOD, pp. 371–382 (2004)