Interactive Generation of Integrated Schemas

Laura Chiticariu[*]
UC Santa Cruz
laura@cs.ucsc.edu

Phokion G. Kolaitis[†]
IBM Almaden
kolaitis@almaden.ibm.com

Lucian Popa[‡]
IBM Almaden
lucian@almaden.ibm.com

ABSTRACT

Schema integration is the problem of creating a unified target schema based on a set of existing source schemas that relate to each other via specified correspondences. The unified schema gives a standard representation of the data, thus offering a way to deal with the heterogeneity in the sources. In this paper, we develop a method and a design tool that provide: 1) adaptive enumeration of multiple interesting integrated schemas, and 2) easy-to-use capabilities for refining the enumerated schemas via user interaction. Our method is a departure from previous approaches to schema integration, which do not offer a systematic exploration of the possible integrated schemas.

The method operates at a logical level, where we recast each source schema into a graph of concepts with Has-A relationships. We then identify matching concepts in different graphs by taking into account the correspondences between their attributes. For every pair of matching concepts, we have two choices: merge them into one integrated concept or keep them as separate concepts. We develop an algorithm that can systematically output, without duplication, all possible integrated schemas resulting from the previous choices. For each integrated schema, the algorithm also generates a mapping from the source schemas to the integrated schema that has precise information-preserving properties. Furthermore, we avoid a full enumeration, by allowing users to specify constraints on the merging process, based on the schemas produced so far. These constraints are then incorporated in the enumeration of the subsequent schemas. The result is an adaptive and interactive enumeration method that significantly reduces the space of alternative schemas, and facilitates the selection of the final integrated schema.

Categories and Subject Descriptors

H.2.1 [**Logical Design**]: Schema and subschema; H.2.5 [**Heterogeneous Databases**]: Data translation

General Terms

Algorithms, Design

Keywords

Schema integration, model management, schema mapping, data integration, interactive generation, concept graph

1. INTRODUCTION

Schema integration is the problem of creating a unified target schema from a set of existing source schemas that relate to each other, possibly via correspondences between their elements or via some other forms of schema mappings such as constraints or views. By providing a standard representation of the data, the integrated target schema can be viewed as a means for dealing with heterogeneous data sources.

The schema integration problem is encountered in data integration and in several other related contexts. In data integration [12], the unified schema yields a single access point against which queries are posed to access a set of heterogeneous sources. Other applications include consolidating data sources of merged organizations into one database or warehouse, and integrating related application silos to create aggregated intelligence. In general, schema integration is a form of "metadata chaos" reduction: quite often, many overlapping schemas (variations or evolutions of each other) exist, even in the same computer, and need to be consolidated into one. Schema integration is recognized as one of the building blocks for metadata applications in the model management framework of Bernstein [3].

Schema integration is a long-standing research problem [2, 6, 14, 18, 21] and continues to be a challenge in practice. All the approaches that we know require a substantial amount of human feedback during the integration process. Furthermore, the outcome of these approaches is only one integrated schema. In general, however, there can be multiple possible schemas that integrate the data in different ways and each may be valuable in a given scenario. In some of the previous approaches, some of these choices appear implicitly as part of the design process, while interacting with the user. However, there is no principled approach towards the enumeration of these choices. In this paper, we develop a method and a design tool that provide: 1) adaptive enumeration of multiple interesting integrated schemas, and 2) easy-to-use capabilities for refining the enumerated schemas via user interaction. Furthermore, the method operates at a logical, conceptual level that abstracts away the physical details of relational or XML schemas and makes it easy to express user requirements.

Overview of our approach We assume that we are given a set of two or more source schemas (describing data in a common domain) together with a set of *correspondences* that relate pairs of
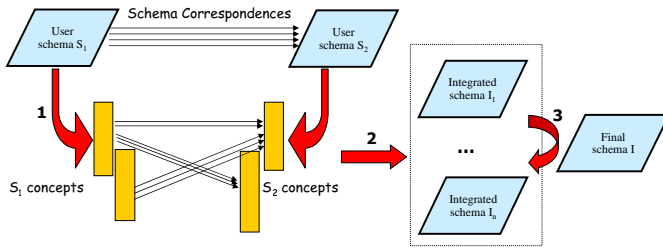
Figure 1: Overview of our method.

elements in these schemas. Correspondences signify "semantically equivalent" elements in two schemas. They can be user-specified or discovered through schema matching techniques [20]. Given such input (source schemas and correspondences), our enumeration approach generates many possible *meaningful* design choices for the integrated target schema. At the same time, we also generate, for each choice of an integrated schema, the *mapping* that specifies how the data in each of the source schemas is to be transformed to the integrated schema.

A high-level overview of how a user can operate our system is schematically illustrated in Figure 1, for the case of two source schemas. We note that we are able to handle any number of source schemas, in general. The architecture is the same except that we require correspondences between multiple pairs of schemas.

As an initial step in our approach (Step 1 in Figure 1), each source schema, with its constraints and nesting, is recast into a higher-level graph of concepts with `HasA` relationships. Each concept is essentially a relation name with an associated set of attributes. A concept intuitively represents one category of data (an entity type) that can exist according to a schema (e.g., "department", "employee", etc.). Concepts in a schema may have references to other concepts in the schema and these references are captured by `HasA` edges (e.g., "employee" contains a `HasA` reference to "department"). For the most part, our subsequent integration method operates at the higher level of abstraction that is offered by the concept graphs.

Next, we identify *matching* concepts in different graphs by taking into account correspondences between their attributes. For every pair of matching concepts we then have the alternative of merging them into one integrated concept or of leaving them as separate concepts.

At the core of our method for exploring the integrated schemas that result from the above choices (Steps 2 and 3 in Figure 1), there are three novel components that we develop. First, we give an algorithm for generating *one* integrated schema, given a *fixed* choice of which matching concepts to merge. We then *define* the space of candidate schemas via an enumeration procedure that takes into account *all* possible merging choices (rather than a fixed one). This enumeration procedure provides the basis for more efficient and more directed ways of exploring the space of candidate schemas. In this paper, we provide one such directed method (our third component) that explores only a selected set of candidate schemas, *interactively, based on user-specified constraints*, ultimately leading to the desired integrated schema.

Generating one integrated schema Given a fixed choice of which concepts to merge, we develop an algorithm (ApplyAssignment) that produces a single integrated schema. This schema preserves, in a precise sense, all the attributes and relationships of the source schemas. The algorithm includes an interactive feature that allows the users to specify how to merge redundant relationships that may arise in the process. At the same time, the algorithm generates a

mapping from the source schemas to the integrated schema that has precise information-preserving properties. The mapping generation component is in the spirit of the more general algorithms of mapping systems such as Clio [17], which construct mappings between independently designed source and target schemas. However, our mapping algorithm is more direct and has no ambiguity, by taking full advantage of how the integrated schema is generated from the source schemas.

Conceptual enumeration of the candidate schemas We develop an *enumeration algorithm* that can systematically generate all possible integrated schemas (and the associated mappings), by considering all possible choices of which concepts to merge. An essential feature of the enumeration algorithm is that it avoids exploring different configurations that yield the same schema. This duplication-free algorithm makes use, as a subroutine, of a *polynomial-delay* algorithm by Creignou and Hébrard [8] for the enumeration of all Boolean vectors satisfying a set of Horn clauses. Polynomial-delay [11] means that the delay between generating any two consecutive outputs (satisfying assignments in this case) is bounded by a polynomial in the size of the input. In a precise sense, this is the best one can hope for when the number of outputs is exponential in the size of the input.

Interactive exploration of the space The users of our tool do not need to see all the candidate schemas. Moreover, the tool does not need to generate (a priori) all the candidate schemas. The full enumeration of all such schemas is not viable, even if duplicates are avoided. Thus, we devise an adaptation of the enumeration algorithm described above so that schemas are generated on demand and in combination with user interaction. Users can browse through the schemas that were generated so far and can also request the generation of the next integrated schema. More interestingly, users can specify *constraints* on the merging process itself, based on the schemas they have already seen. Although constraints are added primarily for semantic reasons (to incorporate the domain knowledge of the expert), they also have the benefit of greatly reducing the space of candidate schemas. Each constraint can cut the space by as much as half. The result is an adaptive enumeration procedure that can quickly converge to a final integrated schema.

User constraints can be given through a visual interface, directly in terms of the concepts and of the matchings between them. For example, the user can enforce a pair (or a group) of matching concepts to be always merged or never merged. Additionally, the user can give constraints that enforce the preservation of certain structural patterns in the input schemas. We show via a series of experiments how user interaction can easily, and in a principled way, narrow down the set of candidate schemas to a few relevant ones.

Furthermore, our experiments show that our algorithm for exploring schemas has also good performance, in the sense that the time to generate the next schema is low (sub-second for all the experiments we ran). Thus, a user of our tool experiences only a small delay before seeing a different schema.

We note that the users of our tool are intended to be domain experts with good familiarity with database and schema design. Thus, they are not end-users but rather developers or data architects.

2. INTEGRATION THROUGH CONCEPTS

In this section, we explain the basic ingredients in our framework, namely the graphs of concepts, and show how the schema integration problem is recast into a problem of merging graphs of concepts.

We first describe the schemas and correspondences that can be input to our method. We shall use the following example throughout the paper. Consider the source schemas S_1 and S_2 depicted
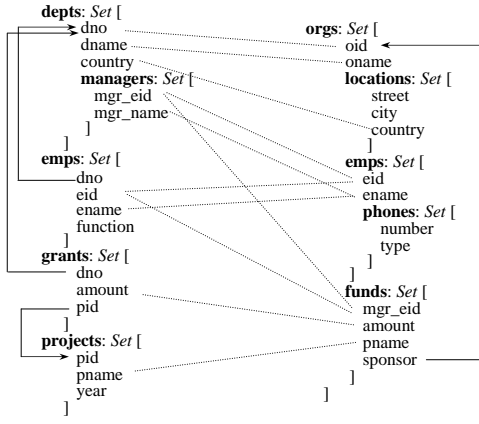
```
depts: Set [                    orgs: Set [
    dno                             oid
    dname                           oname
    country                         locations: Set [
    managers: Set [                     street
        mgr_eid                         city
        mgr_name                        country
    ]                               ]
]                               emps: Set [
emps: Set [                         eid
    dno                             ename
    eid                             phones: Set [
    ename                               number
    function                            type
]                                   ]
grants: Set [                   ]
    dno                         funds: Set [
    amount                          mgr_eid
    pid                             amount
]                                   pname
projects: Set [                     sponsor
    pid                         ]
    pname               ]
    year
]
```

Figure 2: Two input schemas and their correspondences.

in Figure 2. The schemas are shown in a nested relational representation that is used as a common abstraction for both relational and XML schemas. This representation is based on nested sets and records.

Formally, a nested relational schema [17] consists of a set of labels (or *roots*), each with an associated *set type* Set τ, where τ is defined by: $\tau ::= \underline{b} \mid [l_1 : \tau_1, \ldots, l_n : \tau_n] \mid$ Set τ. Here, $\underline{b}$ represents an atomic type such as *string* or *integer*, while $[l_1 : \tau_1, \ldots, l_n : \tau_n]$ is a record type where $l_1, \ldots, l_n$ are labels (or *elements*)[1]. Note that the definition is recursive and sets and records can nest in arbitrary ways. In Figure 2 we do not show any of the atomic types (i.e., the types of the leaf-level elements), for simplicity. We shall often use the term *attributes* for the atomic type elements.

In general, attributes in a source schema are all assumed to have a unique identity (for example, we can use the full path in the schema, for the purpose of identification). However, in examples, we shall often just use the name of the leaf-level element to refer to an attribute, whenever there is no confusion.

The first schema represents departments with their employees and grants, as well as the projects for which grants are awarded. The depts root is a set of department records, each with three atomic components and a set-valued component, managers, which represents a (nested) set of manager records. The arrows in schema $\mathbf{S}_1$ represent foreign key constraints: a grant has references to both a department and a project, while an employee has a reference to a department. The second schema includes a set of organization records with nested sets of locations, employees and funds. The information about managers has been condensed in this schema to one field (mgr_eid in a fund record). Moreover, in this schema, employees have phones and each fund has a sponsoring organization (represented by a foreign key constraint) which may be different from the parent organization.

Figure 2 also shows *correspondences* between atomic type elements (attributes) of the schemas $\mathbf{S}_1$ and $\mathbf{S}_2$. These correspondences are bi-directional and signify "equivalent" attributes (i.e., that can carry the same data) in the two schemas. They can be specified by the user or discovered through schema matching techniques. We only consider correspondences between attributes, since these are the elements that carry actual data. Note that there can be attributes with no correspondences and also attributes with multiple correspondences. For example, mgr_eid in $\mathbf{S}_1$ matches both

mgr_eid and eid in $\mathbf{S}_2$ (possibly with less confidence for the second one; in this paper, we ignore weights on correspondences and, instead, treat all correspondences the same).

Schema integration desiderata Assume that we are given n source schemas $\mathbf{S}_1, \ldots, \mathbf{S}_n$ and a set of correspondences that relate pairs of attributes in these schemas. We would like to compute an *integrated target schema* $\mathbf{T}$ and a *set $\mathcal{M}$ of mappings* from the source schemas to the integrated schema, such that $\mathbf{T}$ and $\mathcal{M}$ satisfy the following informal requirements:

1. The integrated schema $\mathbf{T}$ is capable of representing *all* the atomic-type information in the source schemas, in the sense that every attribute occurring in $\mathbf{S}_1, \ldots, \mathbf{S}_n$ must be represented in $\mathbf{T}$. However, and this will often be the case, it is possible that one target attribute may represent multiple source attributes that are "equivalent" according to the correspondences.

2. The integrated schema $\mathbf{T}$ does not represent any extra atomic-type information not present in the sources, in the sense that every attribute of $\mathbf{T}$ must represent some attribute of $\mathbf{S}_1, \ldots, \mathbf{S}_n$.

3. Every tuple and every join[2] of tuples that can be present in a source database conforming to a source schema is "transferred" via $\mathcal{M}$ into a similar tuple or join of tuples in the target. In a sense, we require the preservation of all the basic relationships that can exist in each of the sources. (All the necessary notions shall be formally defined when we describe our mapping generation component.)

We now embark on the exposition of the main steps towards achieving such integrated schema $\mathbf{T}$ and set $\mathcal{M}$ of mappings.

2.1 Concepts

The first key idea is to abstract the concrete physical layout of schemas into a more logical view that is based on *concepts*. Each schema (with constraints and with nesting) can be replaced by a graph of flat concepts where the edges represent <u>HasA</u> relationships. Formally, a concept graph can be defined (independently of a schema) as follows. We fix $\mathcal{U}$ to be a universe of attributes. We also fix $\mathcal{L}$ to be a universe of labels (different from attributes).

DEFINITION 2.1 (CONCEPT GRAPH). *A concept is a relation name C associated with a subset $\underline{\mathtt{att}}(\mathtt{C})$ of $\mathcal{U}$ (these are the attributes of C). A concept graph is a pair $(V, \underline{\mathtt{HasA}})$ where V is a set of concepts and $\underline{\mathtt{HasA}}$ is a set of directed edges between concepts, such that each edge has a label from $\mathcal{L}$. We write A $\underline{\mathtt{HasA}}$ B $[L]$ whenever there is a $\underline{\mathtt{HasA}}$ edge with label L from concept A to concept B.*

Intuitively, the meaning behind A $\underline{\mathtt{HasA}}$ B $[L]$ is that every instance of concept A has a reference (of type L) to exactly one instance of concept B. The role of the $\underline{\mathtt{HasA}}$ edges is to express that certain concepts cannot exist without other concepts (they *extend* or *depend on* those concepts). Also, another way to view an edge A $\underline{\mathtt{HasA}}$ B $[L]$ is that it represents a many-to-one relationship from A to B: there can be zero or more A instances with references to the same B instance. We also note that, in general, there could be more than one $\underline{\mathtt{HasA}}$ edge between two concepts and, moreover, the graph can have cycles.

To illustrate, we show in Figure 3 two concept graphs that "correspond" to the schemas $\mathbf{S}_1$ and $\mathbf{S}_2$. Although each edge has a label (which we assume is either system generated or given by the user), we sometimes drop the label whenever it is not important. Note that in the case of parallel $\underline{\mathtt{HasA}}$ edges, we always need to display the labels in order to distinguish between them (see, for example, the two $\underline{\mathtt{HasA}}$ edges labeled *sponsor* and *owner* in the second concept graph).

---

[1]This is only a simplified abstraction: choice types, optional and nullable elements are also supported in our system implementation.

[2]Either parent-child type of join or foreign key / key type of join.
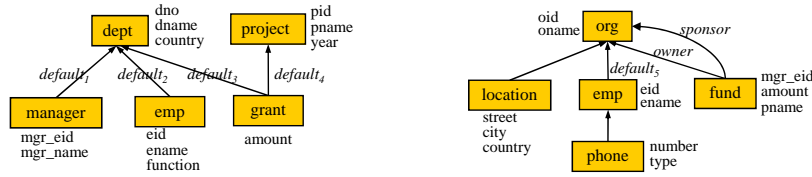
Figure 3: Concept graphs corresponding to schemas S_1 and S_2.

To give the intuition of how these concept graphs relate to the schemas, consider the concept graph for S_1. There, dept and project are top-level concepts (i.e., have no outgoing HasA edges), corresponding to the top-level sets depts and projects. These are standalone concepts that do not depend on anything else. In contrast, there is a HasA edge from manager to dept, since a manager element cannot exist independently of a department (according to the nesting in S_1).

Similarly, there is a HasA edge from emp to dept, reflecting the fact that, in S_1, an employee has a foreign key to a department. Also, grant has edges to both dept and project, since a grant has foreign keys into both department and project. More interestingly, the concept graph for S_2 includes two parallel HasA edges from fund to org, reflecting the fact that a fund is nested under a parent organization (the "owner") and also a fund has a reference to a sponsoring organization (the "sponsor"). Thus, org plays a dual role with respect to fund.

Note that concepts in one concept graph are not intended to represent absolute concepts. For example, location in the second concept graph in Figure 3 represents the "location of an organization" and not a general notion of location. Similarly, emp represents the notion of an "employee within an organization" and not a general employee. Thus, concepts within one concept graph reflect a particular way of modeling the data. However, as schemas are integrated, concepts from different schemas can be merged and, thus, accumulate features (e.g., attributes and relationships) from the different schemas.

2.2 Extracting Concepts from Schemas

In order to extract the concepts and the relationships that are implicit in a schema, we use a simple algorithm that creates one concept for each set-type element in the schema and then uses the structure and the constraints in the schema to establish the relationships (HasA edges) between concepts.

More concretely, for each set-type (collection) element S in the schema, we compute a concept C_S such that: (1) C_S includes all the attributes under S (without attributes from any other set type elements that may be nested under S), (2) C_S has a HasA edge to the concept C_{S_1} encoding the parent collection S_1 (if such parent exists), and (3) C_S has a HasA edge to C_{S_2} whenever C_S has an attribute that is a foreign key referring to one of the attributes (the key) of C_{S_2}. Furthermore, whenever we apply case (3), we drop the foreign key attribute from the list of attributes of C_S (since it is represented at C_{S_2}). In both cases (2) and (3) a fresh label is computed for the HasA edge. We note that case (3) applies also, without much change, when the foreign keys (and keys) are composite.

The concept graphs resulting from the two source schemas S_1 and S_2 are shown in the earlier Figure 3. Note that the name that we give to a concept or to a HasA edge is not essential. A possible choice for concepts is to generate a name based on the set-type element from which the concept is constructed. For HasA edges that are derived from foreign keys, we can use the name of the foreign key attribute.

We also note that it is necessary to keep track of the implicit mapping from a schema to its concept graph. In particular, for each concept, we remember which set type element it corresponds to, and for each HasA edge, we remember either the parent-child relationship or the foreign key constraint it was generated from. This mapping is necessary to be able to translate any subsequent mappings that are expressed in terms of the concepts back in terms of the input schemas.

Finally, the algorithm can deal with cyclic integrity constraints by simply transferring them into cycles over the HasA edges. (For example, the resulting concept graph may include Dept HasA Emp [manager] and Emp HasA Dept [works_for]).

Once the extraction of concepts is achieved, most of the subsequent processing (including user interaction) is performed at the level of concepts and not schemas. This is beneficial since concepts are simpler and also reflect better the logical meaning behind the schemas. Nonetheless, once we integrate the concepts, we can go back to create an integrated schema (see Section 3.3). We also note that our subsequent integration method can take as input arbitrary concept graphs that are not necessarily the result of extraction from schemas. Thus, it can be applied directly to any logical models for as long as they can be expressed as concept graphs with HasA relationships.

2.3 Matching the Concepts

We now show how the input correspondences that are given in terms of schemas are translated into "matching" edges between the concepts that correspond to the schemas. The result of this translation is a *matching graph* that will be the main object of the subsequent processing (i.e., the actual merging and enumeration).

DEFINITION 2.2. *Let S_1 and S_2 be two source schemas and let C be a set of correspondences between attributes of S_1 and S_2. Let A be a concept of S_1 and B be a concept of S_2. We say that A and B match if there is at least one attribute a in A and one attribute b in B such that there is a correspondence at the schema level between attribute a and attribute b.*

Matching concepts will be our candidates for merging. We next define the notion of a matching graph, where the nodes are the concepts (in all the schemas) while the edges indicate matching concepts. Additionally, the matching graph also records the HasA edges, but they will be distinguished from the matching edges.

DEFINITION 2.3 (MATCHING GRAPH). *Let $S_1, \ldots, S_n$ be schemas and let C be a set of correspondences between attributes of these schemas. The matching graph associated with $S_1, \ldots, S_n$ and C is an undirected graph $G = (V, \underline{HasA}, E)$ where:*

- *The set V of nodes is the set of concepts of $S_1, \ldots, S_n$;*
- *The set HasA is the union of the sets of HasA edges obtained from the individual schemas;*
- *The set E of edges contains exactly one edge for each pair of matching concepts in V.*

Figure 4 shows the matching graph G for our example. (Note that for two schemas, the matching graph is a bipartite graph.) For simplicity, we do not show the HasA relationships as edges, to avoid cluttering, but write them as part of the concepts themselves. For example, we add a statement HasA dept to the manager concept
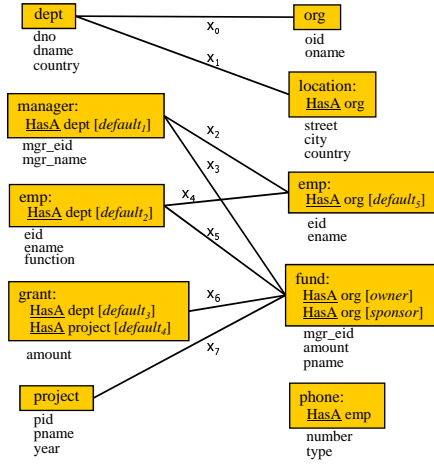
Figure 4: A matching graph.

definition to denote that there is a HasA edge from manager to dept. (Notice that, at this point, a HasA edge never relates concepts that come from different schemas.) In the figure, we aligned concepts in the two schemas so that matching concepts are close to each other. The layout used in the figure is in fact very similar to the way matching concepts are illustrated in the actual tool [7], where we also use specialized graph-displaying packages to help visualizing the matching.

The matching edges in our example are denoted as x_0 to x_7. In general, an edge x between concepts A and B may exist because of multiple pairs of attributes (a, b) that satisfy the condition in Definition 2.2. For example, x_0 exists due to the pairs (dno, oid) and (dname, oname). Nevertheless, we add only one edge between such concepts.

In general, one concept may match with multiple concepts in other schemas. For example, dept matches with org but also with location, since the attribute country in dept has a correspondence to attribute country in location. A priori, we should not assume that dept may match "better" with org than with location. In fact, it may happen that location is meant to represent a branch of an organization, and dept in the first schema also has the meaning of a branch, in which case dept matches better with location. As another example of multiple matchings, the concepts of manager and emp in the first schema match with both emp and fund in the second schema.

The enumeration algorithm (Section 5) will take into account all choices of merging and will explore all possible ways of integration. Note that there may also be concepts that do not match with any concept. For example, phone is specific to the second schema. Even though they have no matches, such concepts are still involved in the merging process, since they may have HasA edges to other concepts.

3. MERGING THE CONCEPTS

Assignments We can obtain different ways of merging the concepts by considering different subsets of edges in the matching graph G. Let A and B be two matching concepts and let x denote the edge between these concepts in G. We can think of the edge x as having a value of 0 or 1: $x = 0$ means that A and B need not be merged (i.e., the edge can be ignored), while $x = 1$ means that A and B must be merged (i.e., the edge must be applied). Every Boolean assignment X to the set E of edges in G will yield an integrated concept graph which in turn will result in an integrated schema. The following are two assignments for the edges in our example:

ApplyAssignment(G, X)
Input: Matching graph $G = (V, \underline{\text{HasA}}, E)$, Boolean assignment X for E.
Output: Integrated concept graph $G' = (V', \underline{\text{HasA}}')$, mapping $\mathcal{M}$ between source and integrated concept graphs.
Let E_X be the subset of edges that have been assigned the value 1.

1. Create the integrated concepts.

   (a) Compute the connected components in the graph $G_X = (V, E_X)$.
   (b) For every connected component $[A_1, \ldots, A_k]$ of G_X, where $A_1, \ldots, A_k$ are concepts in V, create an integrated concept C (i.e., a node in V'). Let $\underline{\text{att}}(C)$ be the union of the attributes in $A_1, \ldots, A_k$, where corresponding attributes are considered duplicates and are represented only once.
   For every A_i among $A_1, \ldots, A_k$, define $f_X(A_i) = C$. Furthermore, for every attribute a of A_i, define $f_X(A_i.a) = C.a*$, where $a*$ is the representative for the group of duplicates of a.

2. Construct $\underline{\text{HasA}}$ edges between integrated concepts. For every $A \underline{\text{HasA}} B\ [L]$ in G, create $f_X(A) \underline{\text{HasA}} f_X(B)\ [L]$ in G' (i.e., an edge in $\underline{\text{HasA}}'$) and define $f_X(A \underline{\text{HasA}} B\ [L]) = f_X(A) \underline{\text{HasA}} f_X(B)\ [L]$.

3. [*Interactive*] Merging of parallel $\underline{\text{HasA}}$ edges and removal of $\underline{\text{HasA}}$ loops in G'. Affects f_X.

4. Create mapping $\mathcal{M} = \text{MapGen}(G, G', f_X)$.

Figure 5: The algorithm ApplyAssignment**.**

$X_1 : \{\, x_1 = x_2 = x_3 = x_5 = 0\, ,\, x_0 = x_4 = x_6 = x_7 = 1\, \}$
$X_2 : \{\, x_0 = x_3 = x_5 = 0\, ,\, x_1 = x_2 = x_4 = x_6 = x_7 = 1\, \}$

The first assignment requires dept in the first schema to be merged with org in the second schema, emp in the first schema to be merged with emp in the second schema, and grant and project in the first schema to be merged with fund in the second schema. Under the second assignment, we must merge dept with location, manager with the two emp concepts, and grant and project with fund.

We shall sometimes identify an assignment X with the subset E_X of edges that have the value 1 under the assignment X.

In this section, we give an algorithm, ApplyAssignment, that takes a matching graph and *one* assignment for the edges, and produces *one* integrated concept graph. At the same time, we also generate the mapping from the source concepts to the integrated concepts that specifies how source data has to be transformed into the integrated data. In Section 5 we shall elaborate on how to enumerate *multiple*, distinct, integration results, by repeatedly invoking ApplyAssignment on different assignments.

3.1 The ApplyAssignment Algorithm

At the high-level, the algorithm, shown in Figure 5, applies first (in Step 1) all the mergings between concepts that are required by the input assignment. Step 2 creates $\underline{\text{HasA}}$ relationships among the integrated concepts, based on the source $\underline{\text{HasA}}$ relationships. The user is then allowed to refine the resulting integrated concept graph in Step 3. In the process, ApplyAssignment maintains an *integration function* f_X (for the given assignment X) that specifies how each individual attribute, concept or $\underline{\text{HasA}}$ edge in a source concept graph relates to a corresponding attribute, concept, and, respectively, *path* of $\underline{\text{HasA}}$ edges, in the integrated concept graph. We shall elaborate on the use of the integration function in Section 3.2, where we explain how we construct the mapping between the source and integrated concept graphs in Step 4 of ApplyAssignment.

3.1.1 Step 1: Connected Components

First, we compute the connected components in the graph $G_X = (V, E_X)$ that is induced from the matching graph G by considering only the edges in X with value 1. For each connected component, an integrated concept is obtained by taking the union of the attributes of the source concepts in that connected component, while at the same time collapsing duplicate attributes. Two attributes a
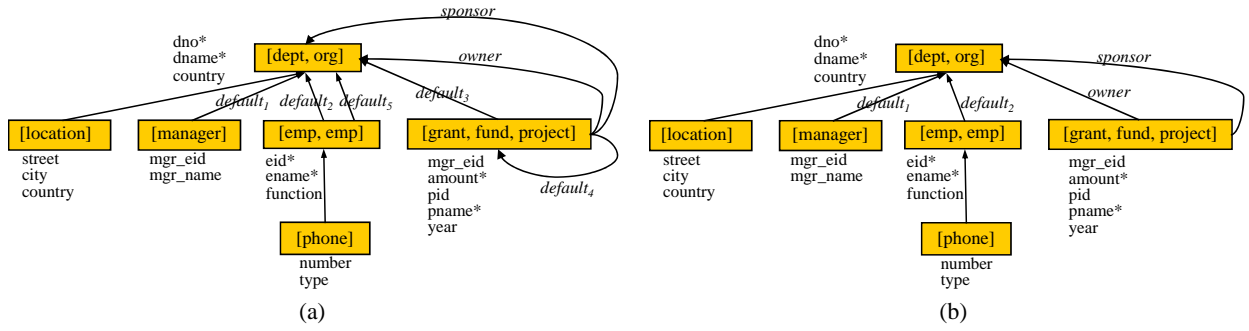
Figure 6: (a) An integrated concept graph after Step 2 of ApplyAssignment**; (b) The integrated concept graph after Step 3.**

and b of source concepts in a connected component are considered *duplicates* if: (1) there exists a correspondence between a and b at the schema level, or (2) there exists another attribute c of a source concept in the same connected component such that c is a duplicate of both a and b. For every group of duplicates we then pick, arbitrarily from the group, a representative that will be subsequently used, in the integrated concept, in place of the individual attributes in the group. As a convention (to signify that a choice was made), we suffix the name of the representative with "*", whenever there are at least two duplicates in the group.

Figure 6(a) shows the integrated concepts that result after Step 1 of ApplyAssignment, when given the earlier assignment X_1. (Ignore the HasA edges between these concepts for now.) There are six integrated concepts, corresponding to the six connected components in the graph $G_{X_1} = (V, E_{X_1})$, where $E_{X_1} = \{x_0, x_4, x_6, x_7\}$. For example, the integrated concept denoted as [dept,org] corresponds to the connected component consisting of the source concepts dept and org. The union of the attributes in dept and org is {dno, dname, country, oid, oname}. However, dno and oid are duplicates and are replaced by a unique occurrence of their representative (chosen as dno*). Similarly, dname and oname are replaced by the representative dname*.

The relationship between the source concepts and the integrated concepts is recorded via the *integration function* f_X. For simplicity, we may write f instead of f_X, if the assignment X is understood from the context. Concretely, each source concept C is mapped into the integrated concept that C is merged into. For example, we have:

$$f(\text{dept}) = f(\text{org}) = [\text{dept,org}], \quad f(\text{location}) = [\text{location}],$$
$$f(\text{grant}) = f(\text{project}) = f(\text{fund}) = [\text{grant,fund,project}], \dots$$

Moreover, each attribute of C is mapped into the representative attribute in $f(C)$. For example, we have:

$$f(\text{dept.dno}) = f(\text{org.oid}) = [\text{dept,org}].\text{dno*},$$
$$f(\text{dept.dname}) = f(\text{org.oname}) = [\text{dept,org}].\text{dname*},$$
$$f(\text{location.country}) = [\text{location}].\text{country}, \dots$$

3.1.2   Step 2: Copying the Relationships

In this step, ApplyAssignment "copies" all source HasA edges into HasA edges on the integrated concepts. Specifically, for each source relationship A HasA B $[L]$, we create the edge $f(A)$ HasA $f(B)$ $[L]$ between the integrated concepts $f(A)$ and $f(B)$. At the same time, we record the correspondence between the source HasA edge and the integrated one as $f(A$ HasA B $[L]) = f(A)$ HasA $f(B)$ $[L]$.

Figure 6(a) shows the integrated concept graph resulting after Step 2 of the algorithm with assignment X_1. We can see that every source HasA edge has a distinct corresponding HasA edge in the

integrated graph. The integration function is now enriched with entries such as:

$f(\text{location HasA dept}) = [\text{location}]$ HasA $[\text{dept,org}]$
$f(\text{emp}_1$ HasA dept $[\text{default}_2]) = [\text{emp}_1, \text{emp}_2]$ HasA $[\text{dept, org}]$ $[\text{default}_2]$
$f(\text{emp}_2$ HasA org $[\text{default}_5]) = [\text{emp}_1, \text{emp}_2]$ HasA $[\text{dept, org}]$ $[\text{default}_5]$

where we write emp_1 for the emp concept in the first schema and emp_2 for the emp concept in the second schema.

As a result of Step 2, the integrated graph may contain parallel HasA edges (see the edges between $[\text{emp}_1,\text{emp}_2]$ and $[\text{dept,org}]$) as well as HasA loops (see the self-loop on $[\text{grant,fund,project}]$). In general, parallel edges must be considered different, since we cannot assume, without additional knowledge, that the relationships encoded by them are the same. For example, we cannot automatically assume that the relationship between an employee and a department that is coming from the first schema is the same as the relationship between an employee and an organization that is coming from the second schema. (See also *owner* and *sponsor* as an example of two parallel edges that represent different relationships.)

A similar argument prevents us from automatically removing loops: the relationship between the grant part of [grant, fund, project] and the project part of [grant, fund, project] that is now implicit in the fact that the two concepts have been merged may not be the same as the original source relationship between grant and project. We may need both, in which case we have to keep the original edge as a loop.

Hence, Step 2 of the algorithm will include by default all the HasA relationships between integrated concepts that can be derived from the source HasA relationships.

3.1.3   Step 3: Removal of Redundant Relationships

This step of ApplyAssignment allows the user to interactively merge parallel edges and remove loops in the integrated graph, whenever the user deems them as redundant. To represent the user feedback, we use a special form of constraints that allows us to remember the information and re-apply it in subsequent invocations of ApplyAssignment (for different assignments). Since these constraints are used to specify redundant information, we call them *redundancy constraints*. These are constraints on the design of the integrated schema (and not constraints on the data).

Parallel HasA edges can be merged by using redundancy constraints of the form:

$$\begin{aligned} &\text{if} \quad f(A) = f(A'), f(B) = f(B') \\ &\text{then} \quad f(A \text{ HasA } B \ [L]) = f(A' \text{ HasA } B' \ [L']) \end{aligned}$$

where A HasA B $[L]$ and A' HasA B' $[L']$ are two source HasA edges. The meaning of such constraint is that, in *any* integrated graph where we merge A and A' (i.e., $f(A) = f(A')$) and we also merge B and B' (i.e., $f(B) = f(B')$), the two parallel HasA edges

with labels L and L' that result in the integrated graph must be considered equal. In a sense, the constraint says that the two source relationships A `HasA` B $[L]$ and A' `HasA` B' $[L']$ are equivalent, whenever the concepts involved in the relationships are merged, pairwise.

To enforce such constraint, one of the two parallel edges must be removed. We take the convention that the first edge will always be removed in favor of the second. As a result of applying the constraint, the integration function f is also updated so that the two source edges map both into the surviving edge in the integrated graph.

For the integrated concept graph in Figure 6(a), a user may state the following constraint:

$(\mathcal{C}_1)$ **if** $f(\mathsf{emp}_2) = f(\mathsf{emp}_1)$, $f(\mathsf{org}) = f(\mathsf{dept})$
       then $f(\mathsf{emp}_2\ \underline{\mathsf{HasA}}\ \mathsf{org}\ [\text{default}_5]) = f(\mathsf{emp}_1\ \mathsf{HasA}\ \mathsf{dept}\ [\text{default}_2])$

to remove the edge from [emp_1,emp_2] to [dept,org] that is labeled $default_5$ in favor of the edge labeled $default_2$ (see Figure 6(b)). A similar constraint can express the merging of the two parallel edges between [grant,fund,project] and [dept,org] that are labeled $default_3$ and $owner$ into one edge labeled $owner$.

Note that such constraints will also apply in other integrated concept graphs, as long as the premises of the constraints are satisfied. For example, assume that later on, the system generates, based on some other assignment, a different integrated graph where emp_1 and emp_2 are still merged (possibly with some other concepts, like manager), and where dept and org are still merged (possibly with some other concepts, like location). Then the user does not have to restate that emp_1 `HasA` dept and emp_2 `HasA` org are the same relationship.

Loops in the integrated graph can be eliminated by means of redundancy constraints of a slightly simpler form:

       if $f(A) = f(B)$
       then $f(A\ \underline{\mathsf{HasA}}\ B\ [L]) = f(A)$

In the above, A `HasA` B $[L]$ is a source `HasA` edge. The meaning of the constraint is that, in *any* integrated graph where we merge A and B (i.e., $f(A) = f(B)$), the resulting loop $f(A)$ `HasA` $f(A)$ [L] must be removed from the integrated graph, and the source `HasA` edge must be represented by $f(A)$.

Based on the integrated concept graph in Figure 6(a), a user may state the following constraint:

$(\mathcal{C}_2)$ **if** $f(\mathsf{grant}) = f(\mathsf{project})$
       then $f(\mathsf{grant}\ \underline{\mathsf{HasA}}\ \mathsf{project}\ [\text{default}_4]) = f(\mathsf{grant})$

to remove the loop labeled $default_4$ (as in Figure 6(b)). As a result, the relationship between grant data and project data is encoded directly within the integrated concept [grant,fund,project] (within one tuple). The integration function is changed accordingly so that the source `HasA` edge from grant to project is mapped into the single concept [grant,fund,project].

From a user interaction point of view, we note that the constraints can be discovered by visualizing the parallel edges or loops that arise in an integrated concept graph. In the visual interface of the tool, the cause for such parallel edges or loops is traced back to the sources (via the integration function f). The user can then immediately state the "desired" constraints in terms of the source edges (e.g., state that two source edges are equivalent, or that one source edge should be collapsed whenever it becomes a loop). We view the mechanism of user constraints (redundancy constraints here, and enumeration constraints later in Section 5) as a form of learning domain knowledge from a user. Once such knowledge is learned, it is automatically reapplied in other configurations, during the same run of the tool.
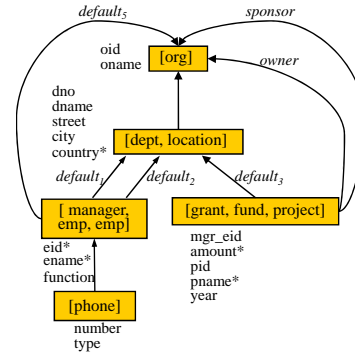

Figure 7: Another integrated concept graph.

3.1.4   *The General Form of Redundancy Constraints*

More generally, we allow users to specify constraints that map a source `HasA` edge into a path of zero, one or *more* `HasA` edges in the integrated concept graph. We have seen examples where a source `HasA` edge is mapped into a path of length zero (e.g., $\mathcal{C}_2$) or of length one one (e.g., $\mathcal{C}_1$). The following example illustrates a case where a source `HasA` edge needs to be mapped into a path of two edges in the integrated graph. For brevity, we shall use the notation $A \xrightarrow{L_1} B \xrightarrow{L_2} C$ to represent a path of edges A `HasA` B $[L_1]$ and B `HasA` C $[L_2]$.

Consider the integrated graph in Figure 7, which is obtained by ApplyAssignment when given the earlier assignment X_2 (instead of X_1) and the constraints in Section 3.1.3. The following constraint:

if $f(\mathsf{emp}_2) = f(\mathsf{emp}_1)$, $f(\mathsf{dept}) = f(\mathsf{location})$

then $f(\mathsf{emp}_2\ \underline{\mathsf{HasA}}\ \mathsf{org}\ [default_5]) = f(\mathsf{emp}_1) \xrightarrow{default_2} f(\mathsf{dept}) \to f(\mathsf{org})$

implies that the edge from [manager,emp_1,emp_2] to [org] that is labeled $default_5$ is made redundant by the path [manager,emp_1,emp_2] $\xrightarrow{default_2}$ [dept,location] $\to$ [org] and can be removed. The constraint asserts that, in general, whenever we merge emp_2 with emp_1, and dept with location, the source relationship emp_2 `HasA` org $[default_5]$ is equivalent to the relationship implied by the two source edges emp_1 `HasA` dept $[default_2]$ and location `HasA` org. Note that the latter source edges are not in the same source schema and do not form a path. However, the condition $f(\mathsf{dept}) = f(\mathsf{location})$ in the premise of the constraint implies that their images (under f) form a path in the integrated graph.

The general definition of redundancy constraints, allowing to map one edge into a path of n edges, where $n \geq 0$, is immediate and we leave it out due to space limitations.

3.2   Mapping Generation

Figure 8 illustrates MapGen, our mapping generation algorithm. MapGen takes as input the matching graph G, the integrated concept graph G' and the integration function f resulting after Step 3 of ApplyAssignment, and outputs a mapping $\mathcal{M}$ between source and integrated concepts. Specifically, for every source concept C, we create a mapping $\mathcal{M}_C$ in $\mathcal{M}$, which specifies how an instance of C, together with all its relationships, is to be transformed into an instance (possibly associated with other instances) of an integrated concept C'. The mappings that we generate are *constraints* in the spirit of schema mapping tools such as Clio [17]; these constraints represent a logical specification of what the lower-level, executable, data migration code needs to implement.

We use our running example to illustrate MapGen and the mappings it produces. Consider the integrated graph from Figure 6(b). In Step 1 of MapGen, we construct a mapping from one single source concept to one single integrated concept. The following is

MapGen(G, G', f)

Input: Matching graph $G = (V, \underline{\text{HasA}}, E)$, integrated concept graph $G' = (V', \underline{\text{HasA}}')$, and integration function f.

Output: Mapping $\mathcal{M}$ from G to G'.

For every source concept C of G do:

1. Initialize $\mathcal{M}_C$ to be

   $$\underline{\text{for}} \; c \; \underline{\text{in}} \; C \; \underline{\text{exists}} \; c' \; \underline{\text{in}} \; C' \; \underline{\text{where}} \bigwedge_{a \in \underline{\text{att}}(\text{C})} (c.a = c'.a^*),$$

   provided that $f(C) = C'$ and $f(C.a) = C'.a^*$, for each $a \in \underline{\text{att}}(\text{C})$. Initialize queue Q = $\{(c \; \underline{\text{in}} \; C, c' \; \underline{\text{in}} \; C')\}$.

2. Extend $\mathcal{M}_C$ by doing a "parallel chase" with $\underline{\text{HasA}}$ edges starting from C. Repeat steps (a)–(b) until Q is empty.

   (a) Take out $(u \; \underline{\text{in}} \; C_i, u' \; \underline{\text{in}} \; C_i')$ from the first position in Q.

   (b) For every edge $C_i \; \underline{\text{HasA}} \; C_j \; [L]$ do:

   - Let $f(C_i \; \underline{\text{HasA}} \; C_j \; [L]) = C_i' \xrightarrow{L_1} D_1 \xrightarrow{L_2} \ldots \xrightarrow{L_n} C_j'$.
   - Chase $u \; \underline{\text{in}} \; C_i$ of $\mathcal{M}_C$ with the edge $C_i \; \underline{\text{HasA}} \; C_j \; [L]$ by adding, in the $\underline{\text{for}}$ clause, a new variable $w \; \underline{\text{in}} \; C_j$ and join condition $u \; \underline{\text{HasA}} \; w \; [L]$.
   - If $n > 0$, chase $u' \; \underline{\text{in}} \; C_i'$ of $\mathcal{M}_C$ with the path $C_i' \xrightarrow{L_1} D_1 \xrightarrow{L_2} \ldots \xrightarrow{L_n} C_j'$. by adding, in the $\underline{\text{exists}}$ clause, a sequence of variables $w_1 \; \underline{\text{in}} \; D_1, \ldots, w_n \; \underline{\text{in}} \; C_j'$, and join conditions $u' \; \underline{\text{HasA}} \; w_1 \; [L_1]$, $\ldots, w_{n-1} \; \underline{\text{HasA}} \; w_n \; [L_n]$.
   - If $n > 0$, add $\bigwedge_{b \in \underline{\text{att}}(\text{C}_j)} (w.b = w_n.b^*)$ to the $\underline{\text{where}}$ clause of $\mathcal{M}_C$, provided that $f(C_j.b) = C_j'.b^*$, for each $b \in \underline{\text{att}}(\text{C}_j)$. Otherwise ($n = 0$ and $C_i' = C_j'$), add $\bigwedge_{b \in \underline{\text{att}}(\text{C}_j)} (w.b = u'.b^*)$ to the $\underline{\text{where}}$ clause of $\mathcal{M}_C$, provided that $f(C_j.b) = C_i'.b^*$, for each $b \in \underline{\text{att}}(\text{C}_j)$.
   - Insert $(w \; \underline{\text{in}} \; C_j, w_n \; \underline{\text{in}} \; C_j')$ (if $n > 0$) or $(w \; \underline{\text{in}} \; C_j, u \; \underline{\text{in}} \; C_i)$ (if $n = 0$) in Q.

Return $\mathcal{M} = \{\mathcal{M}_C \mid C$ is a source concept in $G\}$.

Figure 8: The algorithm MapGen**.**

the mapping $\mathcal{M}_{\text{grant}}$ constructed for the source concept grant:

> $\underline{\text{for}} \; g \; \underline{\text{in}} \;$ grant $\underline{\text{exists}} \; g' \; \underline{\text{in}} \;$ [grant,fund,project]
> $\underline{\text{where}} \; g$.amount $= g'$.amount*

This assertion specifies that for each instance g of grant, there must exist an instance g' of [grant,fund,project] (which is $f(\text{grant})$) where the value for the attribute amount* is copied from the source attribute amount (since $f(\text{grant.amount}) = $ [grant,fund,project].amount*).

Step 2 is the main component of MapGen and its role is to enrich the concept-to-concept mapping established in Step 1 so that it maps *groups of related instances* rather than isolated instances. To illustrate, according to the first source schema, a grant instance is associated with department and project information through the two $\underline{\text{HasA}}$ relationships labeled $default_3$ and $default_4$. In general, it is desirable to carry over such data associations from the source and preserve them in the integrated data. Consequently, $\mathcal{M}_{\text{grant}}$ is extended in Step 2 of MapGen in order to transfer, in the integrated data, all the instances that are directly or transitively associated with a grant instance.

This extension is done by a *parallel chase* in both the $\underline{\text{for}}$ and $\underline{\text{exists}}$ clauses of the mapping, by recursively joining in all the concepts that are related via $\underline{\text{HasA}}$ relationships. At the same time we extend the mapping on the joined concepts by using the function f.

For our example, in one iteration of Step 2(b), for the edge grant $\underline{\text{HasA}}$ dept [$default_3$], we obtain the following updated mapping:

As it can be seen, the $\underline{\text{for}}$ clause is extended by adding a join with the dept concept. We use the notation $g \; \underline{\text{HasA}} \; d$ [$default_3$] to express the join at the *instance* level between g (an instance of grant) and d (an instance of dept). At the same time, we add a corresponding join in the $\underline{\text{exists}}$ clause. We use the fact that $f(\text{grant} \; \underline{\text{HasA}} \;$ dept [$default_3$]) = [grant,fund,project] $\underline{\text{HasA}}$ [dept,org] [owner]. Based on the integration function, we then add to the $\underline{\text{where}}$ clause of $\mathcal{M}_{\text{grant}}$ all the equalities between the attributes of the instances d of dept and d' of [dept,org].

Next, in a second iteration of the same Step 2(b) of the algorithm, $\mathcal{M}_{\text{grant}}$ is extended along the $default_4$ relationship as follows:

> $\underline{\text{for}} \; g \; \underline{\text{in}} \;$ grant, $d \; \underline{\text{in}} \;$ dept, $p \; \underline{\text{in}} \;$ project; $g \; \underline{\text{HasA}} \; d$ [$default_3$], $g \; \underline{\text{HasA}} \; p$ [$default_4$]
> $\underline{\text{exists}} \; g' \; \underline{\text{in}} \;$ [grant,fund,project], $d' \; \underline{\text{in}} \;$ [dept,org]; $g' \; \underline{\text{HasA}} \; d'$ [owner],
> $\underline{\text{where}} \; g$.amount $= g'$.amount* $\underline{\text{and}} \; d$.dno $= d'$.dno* $\underline{\text{and}}$
> d.dname $= d'$.dname* $\underline{\text{and}} \; d$.country $= d'$.country $\underline{\text{and}}$
> p.pid $= g'$.pid $\underline{\text{and}} \; p$.pname $= g'$.pname* $\underline{\text{and}} \; p$.year $= g'$.year

Here, we used the fact that $f(\text{grant} \; \underline{\text{HasA}} \;$ project [$default_4$]) = [grant, fund, project]. While the $\underline{\text{for}}$ clause is extended with a join with project on $default_4$, there is no need for such extension in the $\underline{\text{exists}}$ clause. The same instance g' of [grant,fund,project] that the grant instance g maps into is also used to map the associated project instance p. We only need to add the equalities between the attributes of p and the corresponding attributes of g' (again, using f).

In the next two iterations of Step 2, the algorithm tries to extend $\mathcal{M}_{\text{grant}}$ along any $\underline{\text{HasA}}$ edges that may be outgoing from the dept and project instances that were added to the $\underline{\text{for}}$ clause. However, dept and project are top-level concepts without any such outgoing $\underline{\text{HasA}}$ edges. Hence, Step 2 finishes at this point and $\mathcal{M}_{\text{grant}}$ is completed.

We note that the parallel chase procedure described above does not terminate in the case of cyclic sets of constraints. To ensure that MapGen terminates (and outputs a finite mapping), we add a simple cycle detection condition that avoids further expansion based on $\underline{\text{HasA}}$ edges of concepts that have been expanded before.

Further remarks on mapping generation The parallel chase we use here is a variation on the known chase technique [1]. The chase in the $\underline{\text{for}}$ clause of the mapping is essentially the same as the standard chase. The additional part is that we extend this chase, in parallel, by using the function f, on the $\underline{\text{exists}}$ clause. Therefore, the mappings transfer all the relationships that can exist in the source into corresponding relationships on the integrated schema.

Our mapping generation algorithm is directed by the function f computed in the ApplyAssignment algorithm. This function dictates which concepts map to which concepts and also dictates what join conditions to use. This is in contrast with more general mapping generation algorithms of schema mapping tools such as Clio [17], which construct mappings between independently designed source and target schemas and, as such, have to consider all possible candidate mappings between all pairs of concepts, and with all possible choices of join conditions. The users of such systems would then have to specify which choices to actually use (e.g., a join on $owner$ or one on $sponsor$). In contrast, in our schema integration context, the function f has already encoded in it which concepts and which joins to pick. Hence, mapping generation is more direct.

Although similar to the source-to-target tuple-generating dependencies [17] used for schema-based mappings (i.e., mappings between relational, nested relational, or XML schemas), the mapping constraints described in this section operate at the level of concept graphs rather than schemas. As such, they can potentially have a wider range of applications that go beyond schema-based integration (e.g., they could represent mappings between the objects in two different applications).

3.3 From Concepts to Integrated Schema

From an integrated concept graph we can generate two types of integrated schemas: relational or nested (XML). In the relational version, each concept is implemented in a standard way, as a relation that includes all the attributes of the concept together with an additional key attribute. The HasA edges are then encoded by adding appropriate foreign keys in the concepts that have outgoing HasA edges.

In the nested version, each concept is implemented using a set-type of records containing the attributes of the concept, plus a key attribute. The set-types are then nested, by using the fact that a HasA edge represents a many-to-one relationship. More concretely, if A HasA B $[L]$, then there are zero or more instances of A that each have one reference (of type L) to one instance of B. We can then choose such edge as an "anchor" for nesting: we nest the A's under B's by nesting the set-type element for A under the set-type element for B. The remaining HasA edges, not used for nesting, are then encoded through key / foreign key relationships (as in the relational case).

After creating the integrated schema, we also create the final mapping from the input schemas to the integrated schema. Although the mappings that are generated by MapGen are formulated in terms of concepts, translating such mappings back in terms of the concrete schemas (source and integrated) is straightforward. Essentially, all the HasA joins have to be reformulated in terms of either parent-child navigation or key / foreign key joins (depending on how the relationships are encoded in the concrete schemas).

Mapping generation is the main step towards generating the actual data transformation script (or view) from the sources to the integrated schema. Once the mappings are generated, we can apply any of the existing techniques [10, 4] for compiling the mappings into the run-time queries (XQuery, SQL, XSLT) that are needed to migrate the data.

4. PRESERVATION PROPERTIES

The following proposition summarizes the main features of the ApplyAssignment algorithm (which includes the MapGen algorithm).

PROPOSITION 4.1. *Let $G = (V, \text{HasA}, E)$ be a matching graph and let $G' = (V', \text{HasA}')$ be the integrated graph produced by ApplyAssignment(G,X), for some assignment X to E. Let f be the integration function produced for X.*

1. *Let A be a source concept. Then for every attribute a of A there is an attribute a^* in the integrated concept $f(A)$ such that $f(A.a) = f(A).a^*$. Conversely, let C be an integrated concept. Then for every attribute c of C there is some source concept A and some attribute a of A such that $f(A.a) = C.c$.*

2. *Let A HasA B $[L]$ be a HasA edge in G. Then there is a path of zero or more HasA edges in G' such that $f(A$ HasA B $[L])$ $= f(A) \xrightarrow{L_1} \ldots \xrightarrow{L_n} f(B)$. Conversely, let C HasA D $[L]$ be a HasA edge in G'. Then there is an edge A HasA B $[L]$ in G such that $f(A) = C$, $f(B) = D$ and $f(A$ HasA B $[L]) = C$ HasA D $[L]$.*

3. *Let I be a data instance for the graph of source concepts, and let J be a data instance for the integrated concept graph that is generated by enforcing, in a canonical way, all the mapping constraints produced by MapGen. Moreover, assume that there are no cycles of HasA edges among the source concepts.*

   For every instance t in I of a source concept C, there is a corresponding instance t' of $f(C)$ that is generated in J, such that for each attribute a of t, the value $t.a$ equals $t'.f(a)$. Moreover, whenever such t in I generates a corresponding t' in J, then for

each instance u in I that t refers to (via a HasA edge) there is a corresponding instance u' that is generated in J. Furthermore, t' and u' are related in J by a path of HasA edges that corresponds (via f) to the HasA edge from t to u.

The proposition is an immediate consequence of the way the algorithms ApplyAssignment and MapGen work. The first part states that all the attributes in source concepts are transferred to attributes of integrated concepts and, moreover, there are no "new" attributes in the integrated schema. Thus, the first two informal requirements stated in Section 2 are satisfied in a precise sense. The second part of the proposition states that all the HasA edges between the source concepts are transferred to paths of HasA edges in the integrated schema and, moreover, every HasA edge in the integrated schema comes from an edge (with the same label) in the source schema.

Finally, the third part of the proposition states a stronger preservation property that holds at the data level, and captures the third informal requirement in Section 2. It states that every instance t of a source concept maps to a corresponding integrated instance t'. Moreover, whenever such mapping of t into t' takes place, we also map all the instances related to t into instances related to t'.

Note that the generation of a canonical integrated instance J based on the mapping constraints produced by MapGen is always possible (and in polynomial time).[3] This is due to the fact that the mappings themselves have no cyclicity. However, in the case of cycles of HasA edges among the source concepts, not all the relationships can be preserved. In particular, the paths of HasA edges between source instances can be of unbounded length. Some form of recursive mappings will be needed to preserve such paths.

5. ENUMERATION OF ALTERNATIVES

We now focus on the problem of enumerating all possible assignments X that will result in different integrated concept graphs. We start by discussing the full enumeration algorithm (Sections 5.1 and 5.2), and then we explain how we make this algorithm interactive and adaptive so that only a partial enumeration is needed (Section 5.3).

5.1 Duplicates and Cycles

As we have seen in Section 3, different assignments (such as the earlier X_1 and X_2) encode different ways of merging the input concept graphs and therefore, may give different results for the integration. A naive enumeration algorithm would exhaustively go through 2^n Boolean combinations, if n is the size of X (i.e., the number of matching edges), and then, for each combination, would run the ApplyAssignment algorithm. Besides the potential infeasibility of enumerating a large number of combinations, the naive enumeration algorithm also has the drawback that there may be many assignments that give the same integrated schema (i.e., *duplicate* assignments). A better approach, which we shall follow, is to avoid, from the beginning, the enumeration of duplicate assignments. As a result, a significant portion of the space of assignments can be pruned.

We now describe how duplicates can arise in a naive enumeration algorithm. Recall that, given an assignment X, Step 1 in the ApplyAssignment algorithm merges together all the source concepts that are connected by the edges selected by X. Thus, there is one integrated concept for each connected component in the subgraph of G induced by X. Suppose now that we include one extra edge x in the set of edges that are selected by X. It is then possible that the

---

[3]In fact, as we mentioned in Section 3.3, we can generate concrete queries that efficiently implement the mapping constraints.

effect of this extra edge x is subsumed by edges that are already selected by X. In other words, x specifies that two concepts A and B should be merged but this merge is already a consequence of other selected edges (i.e., there is already a path of edges selected by X that connects A and B). Thus, applying the assignment $X \cup \{x\}$ will result in the same configuration (same connected components) as applying X.

We say that X and Y are *duplicate assignments* whenever the sets of connected components induced by X and Y, respectively, coincide. Duplicate assignments always result in duplicate integrated graphs. Furthermore, duplicate assignments always arise due to cycles in the matching graph $G = (V, \underline{\texttt{HasA}}, E)$.[4] For example, in the earlier argument with the two duplicate assignments X and $X \cup \{x\}$, the extra edge x closes a cycle, since A and B are already connected by edges from X. If G has no cycles then it can be easily shown that there are no duplicate assignments.

To illustrate, consider our earlier matching graph G, shown in Figure 4. The graph consists of 10 concepts, with 8 matching edges, four of which form a cycle: x_2, x_4, x_5, and x_3. It can be seen that if an assignment contains any three edges along this cycle, then adding the fourth edge yields a duplicate assignment. Equivalently, if X is an assignment that contains all four edges in the cycle, then the following assignments are all duplicates of X: $X - \{x_2\}$, $X - \{x_4\}$, $X - \{x_5\}$, $X - \{x_3\}$. Note that this duplication is independent of the assignments to the other edges (not in the cycle). Thus, for each of the $2^4 = 16$ (partial) assignments to the four edges not in the cycle, we will have 5 assignments that have the same effect (thus, four of them are unnecessary). Accordingly, we can count 16 (partial assignments for edges not in the cycle) $\times$ 4 (duplicate partial assignments to edges in the cycle) $= 64$ assignments that should not be considered (out of the $2^8 = 256$ total number of possible assignments). This duplication is even higher if there are more edges outside the cycle. Also, note that duplication happens in this example because of one cycle only.

In general, cycles appear more naturally and with higher frequency in n-way schema integration. In such scenarios, matching concepts will appear in multiple schemas. Furthermore, we may have mappings between multiple pairs of these schemas, thus easily forming cycles in the resulting graph of concepts. This increased number of cycles will result in an even larger number of duplicate assignments. In the experimental section we give synthetic examples of n-way schema integration and further illustrate the impact that cycles have.

5.2 Duplicate-Free Enumeration Algorithm

We now give our algorithm for duplicate-free enumeration of assignments. As we shall show experimentally in Section 6, this algorithm is a significant improvement over the the naive enumeration algorithm. In Section 5.3, we show how to make the algorithm adaptive, by taking user constraints into account. This will further reduce the size of the space that actually needs to be explored.

The main idea behind the algorithm, as alluded to earlier, is to avoid enumerating assignments that contain exactly $k - 1$ edges of a cycle of length k in the matching graph. Each such assignment is a duplicate of the assignment obtained by adding the remaining k-th edge in the cycle. We can formalize the "removal" of the assignments with $k - 1$ edges by imposing a set of constraints that the assignments must satisfy. Specifically, we use a Horn clause of the form

$$x_1 \wedge x_2 \wedge \ldots \wedge x_{k-1} \rightarrow x_k$$

to specify that every assignment that assigns 1 to $x_1, x_2, \ldots, x_{k-1}$

---

[4]Here, the $\underline{\texttt{HasA}}$ edges are not relevant; we mean cycles in (V, E).

CH-Enum(G)

Input: Matching graph $G = (V, \underline{\texttt{HasA}}, E)$
Output: Enumeration of all the distinct integrated graphs (together with their mappings) that can be obtained from G via ApplyAssignment.

1. Compute all cycles of G (considering only the edges in E).

2. Construct a set Γ of Horn clauses as follows: for each cycle $C = x_1, \ldots, x_k$ of edges in E, add the following k Horn clauses to Γ:

$$x_1 \wedge x_2 \wedge \ldots \wedge x_{k-1} \rightarrow x_k$$
$$x_2 \wedge x_3 \wedge \ldots \wedge x_k \rightarrow x_1$$
$$\ldots$$
$$x_k \wedge x_1 \wedge \ldots \wedge x_{k-2} \rightarrow x_{k-1}$$

3. [Creignou & Hébrard]: Generate all satisfying assignments X of Γ.

4. For each assignment X, output the result of ApplyAssignment(G,X).

Figure 9: Duplicate-free enumeration algorithm.

---

must also assign 1 to x_k. We also take into account all "permutations" of such clauses, and we do this for all the cycles in the matching graph.

The net effect of this is that we reduce the problem of duplicate-free enumeration of assignments needed in our schema integration context to the problem of enumerating all the satisfying assignments for a set of Horn clauses. The latter problem has already been studied in the literature, and there is a good algorithm for it. Indeed, Creignou and Hébrard [8] devised a *polynomial-delay* algorithm for generating all Boolean assignments that satisfy a set of Horn clauses. Polynomial-delay [11] means that the delay until the first satisfying assignment is generated, and thereafter the delay between the generation of any two consecutive satisfying assignments, is bounded by a polynomial in the input size (i.e., the total size of the input Horn clauses). This formalizes the notion of a *tractable* algorithm for an enumeration problem with a possibly exponential set of outputs.

Our resulting algorithm (CH-Enum) for duplicate-free enumeration of assignments (and the corresponding integrated concept graphs) is given in Figure 9. Note that, even though our algorithm uses a polynomial-delay algorithm as a subroutine, it is not itself a polynomial-delay algorithm, since, in the worst case, the number of cycles (and, hence, the number of Horn clauses) may be exponential in the size of the matching graph. Nonetheless, as detailed in Section 6, our algorithm performs well on both synthetic and real-life integration scenarios, and clearly outperforms naive enumeration.

We now briefly describe the Creignou & Hébrard procedure itself, as applied to our scenario. Let Γ be a set of Horn clauses as above, and let $x_1, \ldots, x_n$ be the variables that occur in Γ. Given such input, the algorithm proceeds by recursively considering the variables $x_1, \ldots, x_n$ in order, as follows. Initially, all variables are unassigned. In step i ($1 \leq i \leq n$), if the variable x_i is unassigned, then its value is set first to 1. Otherwise, the variable already has a value (see next), and the algorithm continues with step $i + 1$.

When x_i is set to 1, some of the Horn clauses in Γ may now have all 1's in the left-hand side of the implication. For each such Horn clause, the algorithm tries to propagate the value 1 to the variable in the right-hand side of the implication, in an attempt to satisfy the Horn clause. If the variable on the right-hand side is not already assigned a value of 0 (from a previous step), then we set it to 1 (if not already 1) and continue to the next step ($i + 1$). Otherwise, clearly, the clause cannot be satisfied with the current partial assignment. So, we abandon the branch with $x_i = 1$ and set $x_i = 0$. The algorithm continues with step $i + 1$. Whenever a full assignment is found, we output it and then backtrack to the last variable, x_i, that was assigned 1. If no such x_i exists, we are done. Otherwise, we set that variable x_i to 0 and proceed to explore, again, x_{i+1}.

The main advantage of this algorithm (over an exhaustive enumeration of satisfying assignments) is that as soon as a partial assignment is discovered to be unsatisfiable, none of its supersets are further considered. Hence, the algorithm prunes, early in the search, all the assignments that are guaranteed not to satisfy Γ.

5.3 Adaptive Enumeration

Our tool does not enumerate all integrated schemas at once. Instead, target schemas are output one by one and the user is allowed to browse through the schemas generated so far, as well as request the generation of a new schema. As a result of this interaction, after examining a few integrated schemas, a user may gain more insight on the structure of the desired final schema. For example, the user may see things that are "wrong" and should be corrected. Based on this insight, the user can express additional constraints on how concepts should be merged. These constraints are then used to filter the set of schemas generated so far, and, more interestingly, can be incorporated in the enumeration of the subsequent schemas. The result is an adaptive enumeration procedure that significantly reduces the search space with every constraint learned from the user.

User constraints can be given through a visual interface, directly in terms of concepts or of the matching edges between them. In contrast to the redundancy constraints in Section 3.1 which are applied to merge HasA edges between concepts, the constraints we define in this section express conditions on how to merge the concepts themselves. Furthermore, these constraints directly affect the enumeration procedure. Thus, we shall call them *enumeration constraints*.

We allow two types of enumeration constraints. First, the user can require a matching edge x to be always applied or never applied, expressed as Apply(x) and respectively, ¬Apply(x). Whenever such constraint is applied, the space of possible assignments is reduced in half, since one variable is eliminated (set to either 1 or 0). Second, the user can require constraints of the form Merge(A_1, ..., A_n) or ¬Merge(A_1, ..., A_n), where A_1, ..., A_n are arbitrary concepts in the matching graph (some could be from the same schema). These constraints also reduce, significantly, the space of possible assignments.

The meaning of ¬Merge(A_1, ..., A_n) is that, for every pair of distinct concepts A_i and A_j, $i, j \in [1, n]$, A_i and A_j should never be merged. Thus, each A_i will be in a different connected component. Constraints of the form ¬Merge(A_1, ..., A_n) are useful for enforcing certain structural patterns in the integrated schema. As an example, the user can add the constraint ¬Merge(org, location, emp, phone, fund) which essentially requires the structure of the second schema S_2 to be preserved. There is no merging among the five concepts, and their relative structure will stay the same in the integrated schema. However, the concepts of the first schema can be freely merged into the concepts of the second schema, based on the matching edges.

The semantics of Merge(A_1, ..., A_n) is slightly more complex. This is due to the fact that it may not be possible to merge all the n concepts, simply because there may not be enough matching edges. For example, Merge(dept, fund) cannot result in any merging, since there is no matching edge between dept and fund. The semantics we take is that we group the n concepts into connected components based on all the matching edges in the matching graph. If two concepts are in different components, we do not attempt to merge them. However, if two concepts are in the same connected component, we shall only enumerate integrated graphs that keep them connected. For example, the effect of Merge(dept, org, grant, fund, project) is that the subsequent enumeration shall only consider assignments that merge dept and org, and merge grant, fund, and project.

When a new enumeration constraint $\mathcal{F}$ is specified, the already generated schemas that do not satisfy $\mathcal{F}$ are filtered out. More interestingly, we adapt the CH-Enum algorithm so that it only generates, from that point on, only schemas that satisfy $\mathcal{F}$ (and the other existing enumeration constraints). We explain next how this is done. The main idea is to encode the enumeration constraints, whenever possible, with Horn clauses (similar to how we encode cycles in CH-Enum).

If $\mathcal{F}$ is of type Apply(x), where x is a matching edge, we add the Horn clause $\rightarrow x$ to the set of clauses that are considered by CH-Enum, to encode the fact that every assignment must satisfy $x = 1$. In the implementation, if x is unassigned, then we fix $x = 1$ and continue the enumeration procedure. Here, *fixing* the value of x means that x will never be subsequently changed during the enumeration procedure. If x is currently assigned 1, then we proceed with the enumeration until we reach the point when x has to be switched from 1 to 0. At this point, we fix $x = 1$ and backtrack to the variable before x. If x is assigned 0, then we fix $x = 1$ and backtrack to the variable before x. Hence, we eliminate any assignment that has $x = 0$. Similarly, if $\mathcal{F}$ is of type ¬Apply(x), we add the Horn clause $\rightarrow \neg x$ to encode the fact that every assignment must satisfy $x = 0$. The implementation is similar (but complementary) to the case of Apply(x).

If $\mathcal{F}$ is of type ¬Merge(A_1, ..., A_n), then for every two distinct concepts A_i and A_j, $i, j \in [1, n]$ we enforce the constraint ¬Merge(A_i, A_j) as follows. For each simple path $x_1, \ldots, x_k$ between A_i and A_j in the matching graph, we use Horn clauses of the form $x_1 \wedge \ldots \wedge x_{k-1} \rightarrow \neg x_k$ (together with all the permutations). This encodes the fact that every assignment that assigns 1 to any $k - 1$ variables (edges) along the path $x_1, \ldots, x_k$ must assign 0 to the kth variable. Thus, we make sure that no path of edges connects A_i and A_j.

Finally, the case of Merge($A_1, \ldots, A_n$) is trickier. There is no apparent way to encode such constraint, in general, as a set of Horn clauses. Hence, in the implementation, we cannot direct the CH-Enum algorithm to avoid the generation of violating assignments. However, once an assignment is generated, we can check whether it violates the constraint and then discard any such assignment. Although Merge constraints cannot be used to reduce the space explored by CH-Enum, they are still useful from the user interaction point of view.

6. EXPERIMENTAL EVALUATION

We evaluated the performance of our integration method on synthetic schema integration scenarios, as well as on a few real world scenarios. We show that CH-Enum is much faster compared to the naive enumeration and that it scales well with the increasing complexity of schemas and mappings. In particular, the average time to output the next integrated schema, when using CH-Enum, is fairly low. Furthermore, experiments with enumeration constraints indicate that the space of candidate schemas drastically reduces after adding even a small number of constraints. The system is implemented in Java and all experiments are performed on a PC-compatible machine, with two 2.0GHz P4 CPUs and 4Gb RAM, running Linux and JRE 5.0.

6.1 Synthetic Scenarios

The goal of our experiments with synthetic scenarios is to measure the performance of our integration method along three dimensions: 1) the number N of input schemas, 2) the complexity of each schema, measured by its root fanout F (i.e., the number of top level sets) and nesting depth D (i.e., the number of levels of nested sets), and 3) the degree I of interconnection between the input schemas.
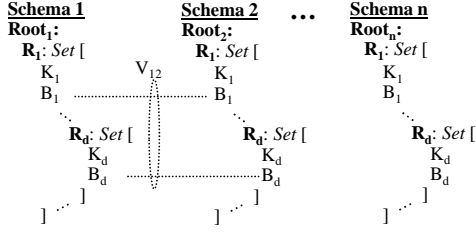
Figure 10: Synthetic schemas used in our experiments.

Figure 10 shows $N = n$ synthetic schemas with root fanout $F = 1$ and nesting depth $D = d$, and a mapping that connects $\mathbf{S}_1$ and $\mathbf{S}_2$. Given input schemas $\mathbf{S}_i$ and $\mathbf{S}_j$, either $\mathbf{S}_i$ and $\mathbf{S}_j$ are not connected, or they are connected through a set of correspondences V_{ij} relating each attribute B_k of $\mathbf{S}_i$ with the attribute B_k of $\mathbf{S}_j$. For $F = f$, the configuration shown in the figure repeats f times (since there are f top-level sets in each schema). The degree I of interconnection between input schemas is the number of schemas $\mathbf{S}_j$ that are connected to an input schema $\mathbf{S}_i$, averaged over all input schemas.

Figure 11(a) shows a comparison in performance between the naive enumeration and CH-Enum. In the experiment we fixed the degree of interconnection to $I = 3$ (i.e., each input schema relates on average to three other schemas). The complexity of each schema was fixed to $D = 3$ and $F = 1$ and we varied the number N of input schemas from 4 to 10. We report the average time to generate the next integrated schema, for both strategies. (In each scenario and for each strategy, we stopped after generating the first 1000 integrated schemas.)

The results show that CH-Enum performs much better than the naive enumeration strategy. For example, in the scenario with $N = 8$ CH-Enum took 2 milliseconds, on average, while the naive algorithm took around 370 milliseconds, to output the next integrated schema. The difference in performance is not unexpected, since the naive algorithm exhaustively enumerates all possible assignments, a large portion of which are duplicate assignments (i.e., leading to the same set of connected components in the matching graph). These duplicates are explicitly removed by checking whether the set of connected components generated in Step 1 of ApplyAssignment has already been encountered. The space needed to store the previously seen sets of connected components is in itself a problem. (In fact, for the scenarios with $N = 9$ and $N = 10$, the naive algorithm ran out of memory, which is why the times are not reported in Figure 11(a).) To illustrate the savings obtained with CH-Enum, consider the scenario with $N = 4$, where there are 3375 distinct integrated schemas. In this scenario, the matching graph consists of 12 concepts and 18 matching edges, and has 21 cycles. The naive enumeration strategy performs duplicate elimination on 2^{18} possible sets of connected components, while CH-Enum directly generates the 3375 distinct ones.

For the rest of the experiments, we report only the times obtained with CH-Enum, since it outperforms the naive enumeration strategy.

In a second experiment, we tested the scalability of CH-Enum with the complexity of the input schemas. Implicitly, this also tests the scalability of the ApplyAssignment algorithm, which is invoked to generate each integrated schema. We used scenarios with two input schemas (i.e., we fixed $N = 2$ and $I = 1$) and generated the first 1000 integrated schemas. Figure 11(b) illustrates the influence of the root fanout (F) and the nesting depth (D) of the input schemas on the average time to output the next integrated schema. As expected, the performance decreases with the increase in the complexity of the schemas.
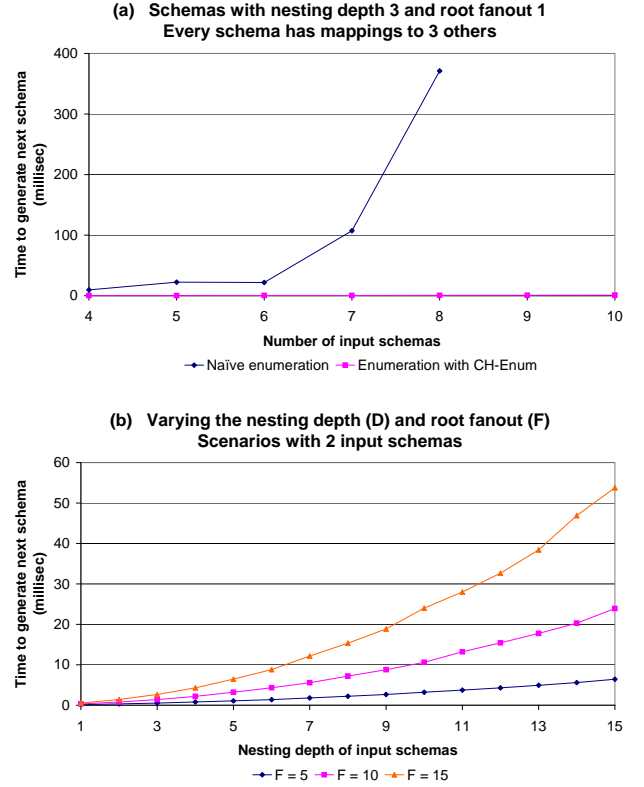

Figure 11: (a) Comparison of CH-Enum with naive enumeration. (b) Time of CH-Enum vs. increasing schema complexity.

6.2 Real Scenarios

We tested the performance and usability of our method in several real-life integration scenarios. The schemas used in these scenarios are: a relational and an XML schema, each representing gene expression experimental results (GENEX); a fragment of the Genomics Unified Schema (GUS) [9] and the BioSQL schema for genomic sequences and features [5]; two XML schemas representing enterprise business objects related to orders, one from SAP and the other one in use with the IBM WebSphere Business Integration (WBI) suite; a relational and the DTD version of the Mondial database [15]; two relational schemas from the Amalgam integration benchmark [13] for bibliographic data; two variations of the XML schema for the DBLP bibliography; the first schema in the Amalgam benchmark and one of the previous DBLP schemas; and three XML schemas, each with a different nesting structure, representing information about departments, projects and employees. Figure 12 shows, for each case, the number of schemas, as well as the number of concepts, matching edges and cycles in the matching graph.

The running times for CH-Enum as well as the size of the space of candidate schemas in each scenario are also shown in Figure 12. CH-Enum performed well in all cases, taking up to 35 milliseconds to generate the next integrated schema, on average. The size of the space of candidate schemas (shown in the Integrated Schemas column) may be large (around or above 1000 schemas in the cases of Mondial, Amalgam, DBLP, Amalgam-DBLP). However, our experiments with user constraints, which we describe next, show that the number of schemas that a user actually explores (and the tool generates) before arriving at the desired schema is much smaller.

The last three columns measure the user interaction effort in

Integration Scenario	Input Schemas	Value Corresp.	Source Concepts	Matching Edges	Cycles	Initialize (sec)	Avg. time /schema (ms)	Integrated Schemas	Integrated schemas after adding the nth enumeration constraint	Enumeration Constraints Enforced	Redundancy Constraints
Genex	2	31	13	6	0	0.014	4.80	64	$1^{st} \Rightarrow 32; 3^{rd} \Rightarrow 8, 6^{th} \Rightarrow 1$	1	4
GUS-BioSQL	2	34	238	9	0	2.667	34.24	512	$1^{st} \Rightarrow 8; 3^{rd} \Rightarrow 2, 4^{th} \Rightarrow 1$	0	6
WBI-SAP	2	46	22	7	0	1.125	8.29	128	$2^{nd} \Rightarrow 8; 3^{rd} \Rightarrow 4, 5^{th} \Rightarrow 1$	1	4
Mondial	2	53	52	19	0	0.056	1.62	>3000	$7^{th} \Rightarrow 1024; 11^{th} \Rightarrow 64; 14^{th} \Rightarrow 8$	2	12
Amalgam	2	30	24	29	3486	92.089	4.83	>3000	$5^{th} \Rightarrow 250; 7^{th} \Rightarrow 10; 9^{th} \Rightarrow 1$	8	0
DBLP	2	18	14	12	10	0.008	27.74	1096	$1^{st} \Rightarrow 144; 3^{rd} \Rightarrow 4; 4^{th} \Rightarrow 1$	1	1
Amalgam-DBLP	2	26	29	10	0	0.014	1.04	1024	$1^{st} \Rightarrow 64; 2^{nd} \Rightarrow 8; 5^{th} \Rightarrow 1$	1	1
Proj-Dept-Emp	3	15	11	10	3	0.005	0.19	250	$2^{nd} \Rightarrow 100; 3^{rd} \Rightarrow 20; 5^{th} \Rightarrow 2$	2	3

Figure 12: Evaluation of CH-Enum on real schema integration scenarios.

Integration Scenario	Enumeration Constraints
GUS-BioSQL	Merge(BioEntry, GOTerm, GOSynonym, Gene, GeneSynonim, Term, TermSynonym) Apply(GORelationship $\leftrightarrow$ TermRelationship) Apply(Taxon$_1$ $\leftrightarrow$ Taxon$_2$) Apply(TaxonName$_1$ $\leftrightarrow$ TaxonName$_2$)
WBI-SAP	$\neg$Apply(Address $\leftrightarrow$ SAP_Order) Merge(Order, PaymentInformation, SAP_Order, SAP_OrderDateData) Apply(OrderLineItem $\leftrightarrow$ SAP_OrderLineItem) Apply(Adjustment $\leftrightarrow$ SAP_OrderLinePricing) Apply(DeliverySchedule $\leftrightarrow$ SAP_ScheduleLines)
Amalgam-DBLP	$\neg$Merge(masterthesis$_1$, phdthesis$_1$, author$_1$, techreport$_1$, book$_1$) Merge(author$_1$, author$_2$, author$_3$, author$_4$) Apply(article$_1$ $\leftrightarrow$ article$_2$) Apply(inproceedings$_1$ $\leftrightarrow$ inproceedings$_2$) Apply(book$_1$ $\leftrightarrow$ book$_2$)
Proj-Dept-Emp	$\neg$Apply(dept$_1$ $\leftrightarrow$ project$_2$) $\neg$Apply(project$_2$ $\leftrightarrow$ dept$_3$) Merge(emp$_1$, emp$_2$, emp$_3$) Apply(dependent$_2$ $\leftrightarrow$ dependent$_3$) Merge(project$_1$, project$_2$, project$_3$)

Table 1: Some of the enumeration constraints used in experiments with real schema integration scenarios.

terms of the number of enumeration and redundancy constraints that need to be added. In the case of enumeration constraints, we show the impact that such constraints have on the overall convergence of our schema integration method. Specifically, the space of remaining candidate schemas (i.e., the space of schemas that satisfy the enumeration constraints added so far and, hence, are of interest to the user) is significantly pruned after adding just a few enumeration constraints. In the DBLP scenario, for example, the number of schemas of interest is decreased from 1096 to 144 after adding the first constraint, then further decreased to 4 after the third constraint, and then decreased to just one after adding the fourth constraint. In the figure, these facts are denoted as $1^{st} \Rightarrow 144; 3^{rd} \Rightarrow 4; 4^{th} \Rightarrow 1$.

To illustrate the enumeration constraints that were added, the following are the four enumeration constraints for the DBLP scenario: 1) do not merge concepts in the second DBLP schema (thus, the second DBLP schema is taken as a reference schema and the other schema is merged into it), 2) merge Article with Pub, 3) merge Inproceedings with Pub, and 4) merge Author (in the first schema) with Author (in the second schema). For Mondial, we used enumeration constraints to enforce the merging of matching concepts such as Sea, Mountain, Lake and others, occurring in the two schemas. As another example, just four enumeration constraints sufficed to reduce the space of candidate schemas to only two integrated schemas of interest in the WBI-SAP scenario. The enumeration constraints

we have added in the WBI-SAP scenario, as well as in some of the other scenarios, are shown in Table 1.

We further note that the number of enumeration constraints required to prune the search space to a few schemas of interest can be seen as a pessimistic upper bound on the number of enumeration constraints that the user has to actually enforce *in practice*. This is because the tool may generate and display the "right" integrated schema much earlier in the process. The reason for this behavior is that the implementation of the enumeration algorithm gives priority to the most merged candidate schemas, which are often what a user wants, provided that other constraints are satisfied. Thus generating schemas from the most merged ones to the least merged ones (due to the fact that we explore assignments starting from all 1's down to all 0's) is quite beneficial in practice.

To illustrate this behavior, the second to last column in Figure 12 shows the number of constraints we have actually enforced before our tool displays the right integrated schema, in each scenario. In the WBI-SAP scenario, for example, the desired integrated schema was generated and displayed after we added the first enumeration constraint, although 5 such constraints are (theoretically) necessary to reduce the search space to this integrated schema. In this scenario, as well as in several others (e.g., Genex, GUS-BioSQL, DBLP), most or all of the desired merging is performed by our tool automatically, and the user needs to enforce at most one constraint to indicate, through $\neg$Apply and $\neg$Merge constraints, which merging choices are undesirable.

Furthermore, as shown in the last column, at most 12 redundancy constraints were needed, over all scenarios, to remove the redundant HasA edges in the integrated schema. As an example, in the DBLP scenario, only one redundancy constraint was needed to merge two parallel relationships "copied" from Author HasA Article and respectively, Author HasA Inproceedings. Finally, we note that in the case of GUS-BioSQL, all that is needed is 4 enumeration constraints (at most) and 6 redundancy constraints to arrive at the merge of two complex schemas of real-life significance to biologists.[5] A similar comment applies to the WBI-SAP scenario, where the input schemas are also quite complex (the concepts have tens of attributes) and are of real-life significance to enterprise applications.

As a note on the implementation of our method, we observe that the number of cycles in the matching graph impacts the initialization time of CH-Enum (i.e., the time to compute all cycles in Step 1). In the Amalgam scenario (our worst case), it takes 92 seconds to compute a total of 3,486 cycles. In general, a large number of cycles in the matching graph constitutes a potential problem for Step 1 of CH-Enum. In the implementation, we can use a "hybrid" approach between the naive enumeration and CH-Enum that limits

---

[5]Of course, generating the correct correspondences between the two schemas is a prerequisite to schema integration.

the number of cycles computed in Step 1, and checks for duplicate assignments as an extra step (since duplicate assignments may still appear due to cycles not found in Step 1). Thus, we bound the initialization time, at the expense of an increase in the time to generate the next schema.

7. RELATED WORK

The distinguishing feature of our approach when compared to existing work on schema integration, model merging, and ontology merging, is the systematic enumeration and exploration of the different integration designs. The enumeration of alternative designs is based on the recognition that correspondences between concepts signify overlap in semantics rather than equivalence, and therefore such concepts may or may not be merged, depending on the scenario. Beyond enumeration, there are several other differences and similarities with the existing work that are worth noting. We focus our discussion on the model merging method of Pottinger and Bernstein [18], since this subsumes much of the earlier work on schema integration [2, 6, 21] and also includes *merging-specific* features that are present in PROMPT [16] and other ontology merging systems, such as FCA-Merge [22]. We note, in this context, that most ontology integration literature has been primarily focused on the problem of ontology alignment, which is deriving relationships across concepts in different ontologies (see ILIADS [23], for a recent example). In contrast, our focus here is on exploring the alternatives for the structural unification of the redundant concepts or attributes (i.e., the merge phase).

One of the main features in [18] is the use of a mapping "in the middle" that essentially drives the integration of the two input models. The mapping, which can be quite complex, can be seen as a "template" for the integrated model, and must be specified by a user before the actual integration. In contrast, the input to our method is just a set of atomic correspondences which can be discovered by an automatic schema matching tool. The additional user constraints in our method are given as the user explores the available choices that the tool discovers (a "learn-as-you-go" approach, as opposed to knowing or guessing in advance what the outcome should be).

As in [18], our method operates at a logical rather than physical schema level. Our meta-meta-model (using their terminology) is more basic; it includes `HasA` edges as the basic form of relationships, and a simpler form of `Contains` (i.e., concepts contain attributes). Nevertheless, our graphs of concepts can express most of the essential features that appear in schemas or in conceptual models such as ER diagrams or UML class diagrams. The work of [18] is extended in [19] by considering the schema integration problem in the context of source schemas related by GLAV mappings, as opposed to just correspondences between schema attributes. It would be interesting to investigate how our enumeration methodology extends to this context.

Finally, we note that we did not address type and representation (e.g., name) conflicts in this paper; we believe that resolution of such conflicts can be applied as a post-processing step, and is complementary to our basic method.

8. CONCLUDING REMARKS

This paper contains a number of contributions to the study of the schema integration problem. Specifically, we developed and implemented a method that, given a set of heterogeneous source schemas and correspondences between them, systematically enumerates multiple integrated schemas and, at the same time, supports refining the enumerated schemas via user interaction. Our method

is built on a principled approach that first formalizes the problem in terms of concept graphs, reduces it to a graph-theoretic problem, and then takes advantage of a known polynomial-delay algorithm for enumerating the satisfying assignments of Horn formulas.

Our enumeration framework has also a conceptual benefit: it is a way of precisely defining the space of candidate schemas in the context of schema integration. Our adaptive enumeration algorithm based on user constraints is one effective way of exploring this space. Other subsequent exploration techniques can be developed in the future, by exploiting various techniques for schema ranking, either through the use of query workloads or by incorporating weights or probabilities into the matchings between attributes and concepts.

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley Publishing Co, 1995.

[2] C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[3] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, pages 209–220, 2003.

[4] P. A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD*, pages 1–12, 2007.

[5] www.biosql.org/.

[6] P. Buneman, S. B. Davidson, and A. Kosky. Theoretical Aspects of Schema Merging. In *EDBT*, pages 152–167, 1992.

[7] L. Chiticariu, M. A. Hernández, P. G. Kolaitis, and L. Popa. Semi-Automatic Schema Integration in Clio. In *VLDB*, pages 1326–1329, 2007.

[8] N. Creignou and J. Hébrard. On generating all solutions of generalized satisfiability problems. *ITA*, 31(6):499–511, 1997.

[9] www.gusdb.org/.

[10] L. M. Haas, M. A. Hernandez, H. Ho, L. Popa, and M. Roth. Clio Grows Up: From Research Prototype to Industrial Tool. In *SIGMOD*, pages 805–810, 2005.

[11] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[12] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, pages 233–246, 2002.

[13] R. J. Miller, D. Fisla, M. Huang, D. Kymlicka, F. Ku, and V. Lee. The Amalgam schema and data integration test suite. www.cs.toronto.edu/ miller/amalgam, 2001.

[14] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *VLDB*, pages 120–133, 1993.

[15] www.dbis.informatik.uni-goettingen.de/Mondial.

[16] N. F. Noy and M. A. Musen. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *AAAI/IAAI*, pages 450–455, 2000.

[17] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.

[18] R. Pottinger and P. A. Bernstein. Merging Models Based on Given Correspondences. In *VLDB*, pages 826–873, 2003.

[19] R. Pottinger and P. A. Bernstein. Schema Merging and Mapping Creation for Relational Sources. In *EDBT (To appear)*, 2008.

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

[21] S. Spaccapietra and C. Parent. View Integration: A Step Forward in Solving Structural Conflicts. *TKDE*, 6(2):258–274, 1994.

[22] G. Stumme and A. Maedche. FCA-MERGE: Bottom-up merging of ontologies. In *IJCAI*, pages 225–234, 2001.

[23] O. Udrea, L. Getoor, and R. J. Miller. Leveraging Data and Structure in Ontology Integration. In *SIGMOD*, pages 449–460, 2007.