STBenchmark: Towards a Benchmark for Mapping Systems

Bogdan Alexe
UC Santa Cruz
abogdan@cs.ucsc.edu

Wang-Chiew Tan
UC Santa Cruz
wctan@cs.ucsc.edu

Yannis Velegrakis*
University of Trento
velgias@disi.unitn.eu

ABSTRACT

A fundamental problem in information integration is to precisely specify the relationships, called mappings, between schemas. Designing mappings is a time-consuming process. To alleviate this problem, many mapping systems have been developed to assist the design of mappings. However, a benchmark for comparing and evaluating these systems has not yet been developed.

We present STBenchmark, a solution towards a much needed benchmark for mapping systems. We first describe the challenges that are unique to the development of benchmarks for mapping systems. After this, we describe the three components of STBenchmark: (1) a basic suite of mapping scenarios that we believe represents a minimum set of transformations that should be readily supported by any mapping system, (2) a mapping scenario generator as well as an instance generator that can produce complex mapping scenarios and, respectively, instances of varying sizes of a given schema, (3) a simple usability model that can be used as a first-cut measure on the ease of use of a mapping system. We use STBenchmark to evaluate four mapping systems and report our results, as well as describe some interesting observations.

1. INTRODUCTION

A fundamental problem in information integration is to precisely specify the relationships, called *mappings*, between schemas. A mapping is a precise specification of how data stored under different representations are related. Mappings are fundamental building blocks in many applications such as data integration, data exchange and peer data management systems [21, 24, 26]. However, specifying mappings (also referred to as the *data programmability problem* in [3]) is a time-consuming and laborious process [19] because schemas are typically heterogeneous, designed independently, in different formats and with different applications in mind. In fact, one of the goals of model management [3] is to make the task of designing mappings between different representations easier.

In this paper, a *mapping system* is a visual programming system with the goal of assisting a mapping designer towards the generation of a precise specification of the relationships between two schemas with less effort. The precise specification is typically described in some programming language (e.g., XSLT or XQuery) and it spells out the data exchange process, i.e., how an instance over one schema, called the source schema, is to be translated into an instance over the other schema, called the target schema. Today, many mapping systems such as Altova Mapforce [29], IBM Rational Data Architect [23], Microsoft BizTalk Mapper which is embedded in Microsoft Visual Studio [45], Stylus Studio [41], BEA AquaLogic [13], and the research prototypes Clio [20] and HePToX [9] have been developed to alleviate the task of designing mappings. Despite the availability of many mapping systems, there has been no benchmark developed for comparing and evaluating them. Similar to the motivation of benchmarking relational database management systems, a benchmark for mapping systems is important for assessing their relative merits, which is in turn important to customers for making the right investment decisions. In fact, a recent workshop on information integration [6] has also raised the need for developing a benchmark for data exchange systems. However, unlike benchmarks for relational database management systems [42] or XML query engines such as [8, 12, 37, 39, 46], it is considerably more challenging to design a benchmark for mapping systems. One major difficulty arises from the fact that there does not exist a standard input language or input methodology for mapping systems. In contrast, benchmarks for RDBMSs and XML query processing systems could leverage their respective standard query languages, SQL and XQuery respectively, to specify their benchmark test cases. We shall elaborate more on the challenges in the design of a benchmark for mapping systems, our solutions and the goals of STBenchmark in Sec. 2.

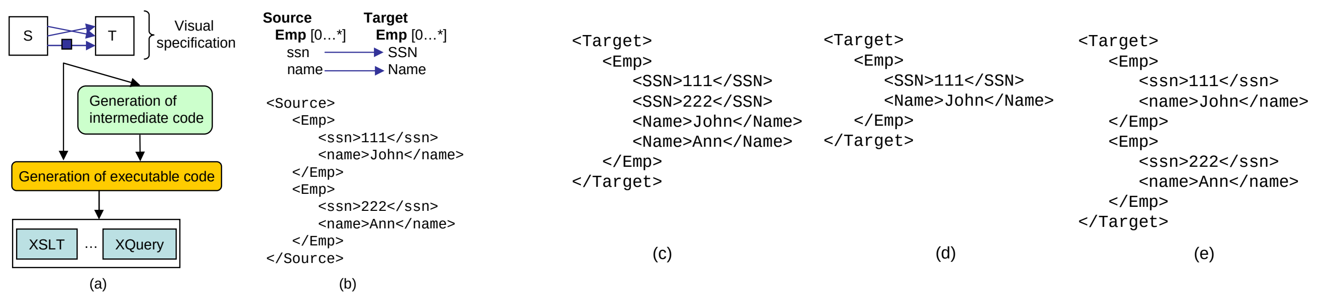
In this paper we present STBenchmark[1][2], a first attempt towards a benchmark for comparing and evaluating mapping systems. Our evaluation criteria are based on the effort needed to implement a mapping through the visual interface of a mapping system, the degree of support offered by a mapping system for the implementation of various scenarios, and the scalability of the generated transformation code in terms of schema and instance sizes. Our specific contributions are the following:

(1) We identify the not obvious challenges that are unique to the development of benchmarks for mappings systems, and explain how STBenchmark deals with these challenges. (see Sec. 2)

(2) We decribe a suite of basic mapping scenarios we believe represents a minimum set of transformations that should be supported through the visual interface of any mapping system. This means that for every basic mapping scenario, the designer should be able to obtain the desired executable code through the visual interface of a mapping system without having to understand and manually modify the executable code. These mapping scenarios are the result of a a careful analysis of constructs commonly needed across different in-

*Work partly done while visiting UC Santa Cruz

[1]standing for Source-to-Target mapping Benchmark
[2]http://www.stbenchmark.org



```
<Source>
   <Emp>
      <ssn>111</ssn>
      <name>John</name>
   </Emp>
   <Emp>
      <ssn>222</ssn>
      <name>Ann</name>
   </Emp>
</Source>
```

```
<Target>
   <Emp>
      <SSN>111</SSN>
      <SSN>222</SSN>
      <Name>John</Name>
      <Name>Ann</Name>
   </Emp>
</Target>
```

```
<Target>
   <Emp>
      <SSN>111</SSN>
      <Name>John</Name>
   </Emp>
</Target>
```

```
<Target>
   <Emp>
      <ssn>111</ssn>
      <name>John</name>
   </Emp>
   <Emp>
      <ssn>222</ssn>
      <name>Ann</name>
   </Emp>
</Target>
```

Figure 1: (a) Architecture of mapping systems. (b) An example visual specification and source instance. (c,d,e) Target instances.

formation integration applications, such as data exchange, data warehouses, XML publishing, schema evolution, as well as real-world mapping specifications. (see Sec. 3)

(3) We present a mapping scenario generator and an instance generator we have designed and implemented. (see Sec. 4.) The mapping scenario generator is able to produce complex mapping scenarios between two schemas and the instance generator is able to construct instances of varying sizes that conform to these schemas. As one can extract schemas from the generated mapping scenarios the mapping scenario generator can also be used as a schema generator. We also describe how our generators can be applied to stress test algorithms that were developed for a wide variety of information integration projects such as schema integration, evolution, as well as composition and debugging of mappings.

(4) We describe a simple usability model, for assessing the ease of use of the visual interface of a mapping system. While it is not meant to replace a much needed comprehensive human-computer interaction study (which is not the subject of this paper) on the usability of mapping systems, it could be used for providing a first-cut measure on the ease of use of a mapping system. (see Sec. 5)

(5) We evaluate four mapping systems with STBenchmark, report our findings and describe some interesting observations (see Sec. 6).

2. CHALLENGES AND SOLUTIONS

State-Of-The-Art in Mapping Systems. Recall that a *mapping system* is a visual programming system that is typically built with the goal of assisting a mapping designer towards the generation of mappings between two schemas with less effort. The typical approach taken by mapping systems to achieve this goal is to use a graphical user interface and a graphical representation that abstracts the underlying specification between two schemas.

Our benchmark is targeted at *relationship-based mapping systems* [36], which adopt the following methodology towards the design of a mapping (see Fig. 1(a)): In the visual interface, one schema (the source schema **S**) is displayed on the left panel of the screen while the other schema (the target schema **T**) is displayed on the right of the screen. After this, the mapping designer is allowed to relate elements of the two schemas by creating lines between them. Some mapping systems [5, 20, 29] can also suggest element correspondences automatically between the two schemas. It is also possible to visually specify complex relationships between elements of the two schemas through a library of supported functions. In some mapping systems, the functions in the library are direct correspondences to functions native to the language in which the transformation is expressed. A function is typically depicted as a box with incoming and outgoing lines that connect to schema elements or other functions. Hence, the *visual specification* that illustrates the source and target schemas, as well as the lines and boxes across elements of the two schemas, provide an intuitive description of the underlying precise specification. An example of a visual specification is shown on top of Fig. 1(b). The visual specification can usually be compiled into different executable languages, such as XSLT, XQuery, Java or C. In tools such as [9, 20], this visual specification is first compiled into an intermediate code from which executable code is generated. Most mapping systems are also able to save the visual specification, either in some proprietary format or directly as executable code, so that the visual specification can be reloaded into the mapping system at a later time. Hence, the input to these mapping systems could either be a visual specification or a file in some proprietary format. It is also worth noting that different mapping systems have different levels of language support. For example, Stylus Studio supports the use of many XSLT functions through its graphical user interface but Clio does not.

Mapping systems are not to be confused with *schema matching systems*. The latter is concerned with obtaining a set of *mapping elements*, where each mapping element indicates how elements of one schema relate to elements of the other schema [35]. The relationship specified between sets of elements could be as simple as stating that the two sets are related, or it could involve a mapping expression such as the concatenation of firstname and lastname of one schema being equal to the name element of the other schema. The set of mapping elements is required as input to the code generation process. Some mapping systems have a matching module that (semi-)automatically derives these mapping elements, while in many others, the set of mapping elements are manually specified through the visual interface. We emphasize that STBenchmark is *not* a benchmark for schema matching systems. There are already proposals for schema matching benchmarks [17, 47]. However, it would be interesting to consider incorporating a schema matching benchmark into STBenchmark in future.

Challenges and Solutions of STBenchmark. A benchmark is "a standardized problem or test that serves as a basis for evaluation or comparison (as of computer system performance)" [30]. Hence, one goal of STBenchmark is to provide a standard set of test cases that could be used to evaluate and compare different mapping systems. However, there are many factors unique to mapping systems that make the design of a benchmark for such systems considerably more challenging than benchmarks for other types of systems such as query engines.

One major difficulty is that currently, there does not exist a standard approach for designing a mapping between two schemas across different mapping systems. Although lines and boxes are visual metaphors commonly used as input across different mapping systems, they are interpreted in distinct ways by different systems. For example, the simple visual specification depicted on top of Fig. 1(b) which consists of a source and target schema with two lines that connect the ssn elements and the name elements, respectively, is compiled into inequivalent XSLT scripts by different mapping systems. The XSLT script that is generated by Altova Mapforce [29] groups all ssns of employees in the source, followed by all names of employees in the source, under a single ⟨Emp⟩ tag. (see Fig. 1(c) which is the result of applying Mapforce's XSLT script on the source instance shown at the bottom of Fig. 1(b).) Thus, in the case when the source instance consists of more than one employee, the target instance generated by the XSLT script does not even conform to the target schema. (In fact, to specify a transformation that copies the source instance, Mapforce requires an additional line between the Emp elements in Fig. 1(b).) The XSLT script that is generated by Stylus Studio [41] creates a single $\langle$Emp$\rangle$ tag within which there is a single $\langle$SSN$\rangle$ and $\langle$Name$\rangle$ tag. Only the first ssn and name of employees in the source are listed under the $\langle$SSN$\rangle$ and $\langle$Name$\rangle$ tag respectively. (see Fig. 1(d).) Microsoft's BizTalk Mapper [45], IBM Rational Data Architect [23] and Clio [20] generate XSLT scripts that return a copy of the source instance. (see Fig. 1(e).)

Since mapping systems interpret the same visual specification in different ways, it is therefore impossible to specify our benchmark test cases as visual specifications to mapping systems. In contrast, observe that benchmarks such as TPC-H [42] are able to leverage the industry-wide standard query language SQL for specifying the input test cases to different database vendors. Our solution to this challenge is to specify each test case as a *mapping scenario* instead.

DEFINITION 2.1. A *mapping scenario* is a triple $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ is a source schema, $\mathbf{T}$ is a target schema, and $\mathcal{P}$ is a precise transformation function on how an instance of $\mathbf{S}$ is to be transformed into an instance of $\mathbf{T}$.

Since mapping systems do not take mapping scenarios as input, each mapping scenario needs to be implemented by the benchmark user by using the visual interface of each mapping system.

An important measure of the quality of a mapping system is its performance. There are two natural aspects of performance: The first is the time the mapping system takes to compile a visual specification into executable code. The second is the performance of the generated executable code. Since all the mapping systems that we encountered do not provide methods by which we can record the time they take to generate executable code, we omit the first performance aspect in STBenchmark. For the second aspect, our solution is to measure the performance of the executable code on a common execution engine. In this paper, we measure the quality of XSLT scripts that are generated by different mapping systems on various mapping scenarios by executing the XSLT scripts on a common XSLT execution engine. We measure how well the generated scripts scale with the size of the mapping scenario and source instance through the use of STBenchmark's mapping scenario and instance generator (see Sec. 4). We focus on XSLT because all mapping systems that we encountered support the generation of executable code in XSLT. Similar tests could be carried out in the future for other executable code generated in other languages.

As described earlier, a mapping system compiles a visual specification into executable languages, such as XSLT, XQuery, Java or C. Since visual interfaces are very much part of mapping systems, it is also important to assess the degree and ease by which one can implement the mapping scenarios with the visual interface of the mapping system. While it is sometimes easy to determine whether a mapping system is able to implement a mapping scenario through its visual interface, the ease by which the mapping scenario can be implemented with the visual interface of the mapping system is significantly harder to quantify. Our solution to this challenge is a simple usability model for assessing the ease of use of a mapping system (see Sec. 5). We believe that our simple usability model is useful for providing a first-cut measure on the ease of use of a mapping system. However, we emphasize that the simple usability model is not meant to replace a much needed comprehensive human-computer interaction study (which is not the subject of this paper) on the usability of mapping systems.

3. BASIC MAPPING SCENARIOS AND REAL SOURCE INSTANCES

In this section, we describe the first component of STBenchmark which consists of a set of basic mapping scenarios that we believe represents a minimum set of transformation functions that should be supported through the visual interface of any mapping system. This means that each basic mapping scenario should be readily implementable through the visual interface of any mapping system without having the mapping designer understand and modify the underlying executable code in order to achieve the desired effect. Thus, valuable designer effort and time could be saved. The basic scenarios are similar in spirit to benchmarks like TPC-H [42] and XMark [39], which consist of a set of queries that is expected to be executable by any database system or XML query engine. Our basic mapping scenarios capture some transformations typical of information integration applications and they are derived by a careful analysis of elementary constructs needed for applications such as data exchange or data warehouses, XML publishing, schema evolution, real-world mapping specifications, as well as the authors' experience in these areas. We emphasize that our basic mapping scenarios are not meant to be an exhaustive representation of all possible transformations. Rather, they are intended to capture common transformation cases that occur in practice and have wide industry relevance. However, they cover all the cases presented in [27].

Every basic mapping scenario is accompanied by a source instance which has been extracted from a real data source such as the BioWarehouse [7] or the DBLP Server [16]. Thus, the transformation scripts generated by a mapping system can be evaluated against these "real" source instances. In what follows, we describe and justify each basic mapping scenario. The XQueries describing the transformation functions can be found in Appendix A.

Copying In many data transformation applications it is frequently the case that an instance or subinstance of a source schema is simply copied to the target.

The source and target schema are shown in Fig. 2(a). The source schema consists of a set of Source/Protein records[3] which consist of three subelements. This set of records is copied to the set of Target/Protein records. □

Constant Value Generation This mapping scenario represents the situation when some constants need to be created in the target. Such scenario occurs frequently in practice according to our experience as well as [7, 14, 27], where constant values that are independent of the source instance need to be added to the target instance.

The target schema is shown in Fig. 2(b). The source schema can be any schema and is therefore omitted. Here, a Target/DataSet element is created to record the name of the database and the date when the database was created. In this case, the constants "SwissProt" and "July 4th" are the Name and LoadingDate respectively. □

Horizontal Partitioning This mapping scenario represents the situation where the contents of a source are partitioned into two or more fragments in the target. This scenario occurs frequently in schema evolution [27]. It may happen that a set of elements grows so large over time that it affects the performance of a database management system. Hence, the database needs to be partitioned. As described below, horizontal partitioning typically requires data filtering support (e.g., selection conditions) in mapping systems.

The source and target schema are shown in Fig. 2(c). The source schema consists of a set of Source/Gene records, which consist of three subelements. The target schema consists of two sets of Target/Gene and Target/Synonym records respectively. (Ignore the WID subelement in the target for this scenario.) The source records

[3]We use the nested relational model to interpret the schemas. Hence, the order in which Source/Protein records occur does not matter.

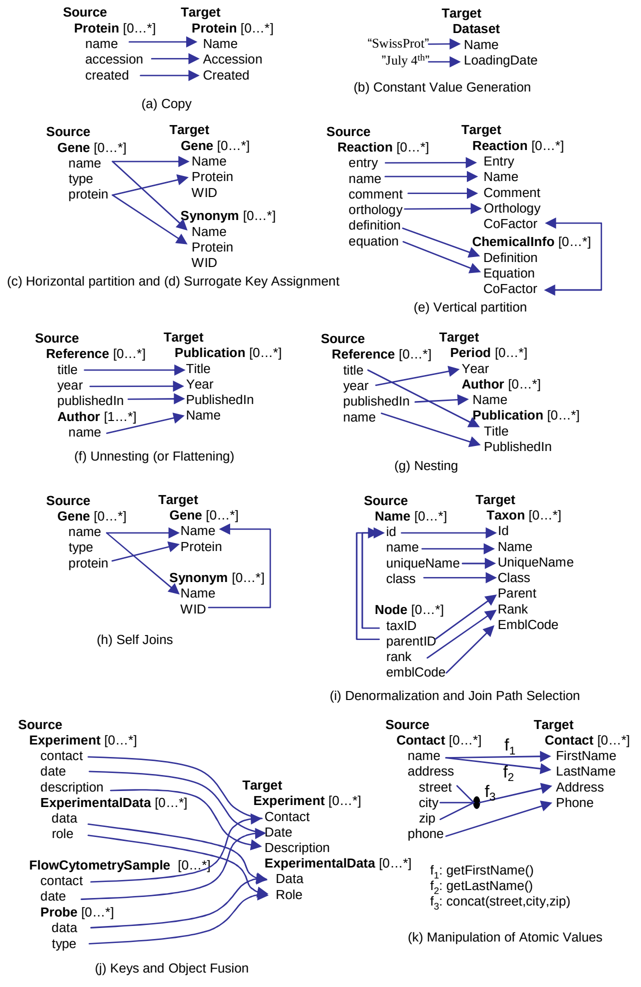


Figure 2: Basic Mapping Scenarios of Sec. 3.

are partitioned horizontally into the two target sets based on the value of the type subelement: Target/Gene consists of Source/Gene records whose type is "primary" and Target/Synonym consists of Source/Gene records whose type is not "primary". It is typically the case that in horizontal partitioning, the schema of each target partition is identical to the source schema. However, we have omitted the "type" subelement in the target partitions in this scenario. □

Surrogate Key Assignment In data warehouses, objects are often given new unique identifiers (or keys) that may be different from the identifiers in the original data sources. The surrogate key assignment scenario depicts this situation.

The source and target schema are identical to those of the horizontal partitioning scenario except that now, we consider the WID subelements in the target schema. See Fig. 2(d). As in the horizontal partitioning scenario, Target/Gene consists of Source/Gene records whose type is "primary" and Target/Synonym consists of Source/Gene records whose type is not "primary". In addition, a fresh new value is generated for every record in the target sets. Target/Gene/WID is the key for Target/Gene records and Target/Synonym/WID is the key for Target/Synonym records. □

Vertical Partitioning The vertical partitioning scenario also occurs frequently when a schema is redesigned or when the schema evolves. In fact, this scenario is also referred to as the normalization task in schema evolution [27].

Unlike horizontal partitioning which splits a set of records into two or more different sets of records in the target based on some condition, vertical partitioning splits a set of records vertically into two or more sets of records in the target. In other words, the target sets are typically projections of the original set of records.

The source and target schema of the vertical partition scenario are shown in Fig. 2(e). The first four subelements and the last two subelements of Source/Reaction are subelements of Target/Reaction and Target/ChemicalInfo respectively. Additionally, Target/Reaction and Target/ChemicalInfo are related via the referential constraint on CoFactor. The transformation takes each record in Source/Reaction and splits it into two smaller records in the target. At the same time, a unique CoFactor value is used to relate the two smaller records in the target sets. □

Unnesting (or Flattening) Another frequently occurring scenario in information integration and particularly in XML-to-relational storage is that of unnesting (or flattening) nested structures.

The source and target schema are shown in Fig. 2(f). The source schema is a set of Source/Reference records within which there is a nested set of Author records. The target schema consists of a set of Target/Publication records without any nested sets. The set of Reference records in the source is flattened into a set of Publication records in the target. In other words, Target/Publication consists of the set of records that is the result of taking the Cartesian product of each Source/Reference record with their respective nested set of Author tuples. □

Nesting The nesting scenario, which is the opposite of unnesting, also occurs frequently in information integration applications such as relational-to-XML publishing [40] and schema evolution [27].

The source and target schema are shown in Fig. 2(g). Source/Reference is a set of "flat" records. In the target, information about references is organized as follows: For each year, there is an associated set of authors who published in that year, and for each author, there is an associated set of publications of that author. □

Self Joins Join is the main operation used to associate information located in different elements. In some cases, the elements to be associated by the join path are located in the same relation. Hence, a relation has to be joined with itself. A classic textbook example is a self-join on the parent relation to retrieve the set of parent-child pairs. Our self-join scenario is similar to the parent-child example and is described below. The scenario has been derived from the real data integration application Biowarehouse [7].

The source and target schema are shown in Fig. 2(h). The source schema consists of a single set of Source/Gene records with three subelements describing the name of the gene, the type (whether it is 'primary' or not) and the protein which the gene belongs to. The target schema consists of two sets of records, Target/Gene and Target/Synonym respectively. Since a protein can have multiple genes but only one of the genes is primary, the primary gene is stored in Target/Gene, while all other genes of the same protein (i.e., synonyms) are stored in Target/Synonyms. Additionally, a foreign key Target/Synonym/WID references Target/Gene/Name to indicate the primary gene of each synonym and the protein to which they collectively belong. □

Denormalization and Join Path Selection Relevant or overlapping information is often found in different data sources and needs to be combined. It may happen that there are multiple ways (i.e., join paths) to combine these data sources. The denormalization and join path selection scenario depicts this situation and can be seen as the reverse of vertical partitioning scenario.

The source and target schema are shown in Fig. 2(i). If a record in Source/Name has id value equal to the taxID value of a record in Source/Node, then the two records are combined to form a Target/Taxon record. Note that Source/Name and Source/Node records can also be joined through the id-parentID subelements. However, in this scenario, we require that Source/Name and Source/Node records be joined through id-taxID subelements. □

Keys and Object Fusion The scenarios presented so far have not taken into consideration any key constraints that may exist in the source or in the target. Keys are frequently used to merge information from different sources that refer to the same real world object (i.e., object fusion) and for duplicate elimination. Such scenarios occur frequently in data integration.

The source and target schema are shown in Fig. 2(j). Every time an experiment is performed, details about the experiment are recorded in Source/Experiment. Each experiment is uniquely identified by the contact and date subelements and may have zero or more experimental data associated with it. An experiment may also be associated with zero or more flow cytometry probes. Information about flow cytometry probes is recorded under Source/FlowCytometrySample. Each FlowCytometrySample is also uniquely identified by its contact and date. Target/Experiment is a warehouse of experimental data and flow cytometry probes. It consolidates the two, based on contact and date, making no distinction between experimental data and probes. In this transformation, we migrate all ExperimentalData from Source/Experiments and all Probes from Source/FlowCytometrySample to Target/Experiment/ExperimentalData. In the target, all ExperimentalData from Source/Experiment and all Probes from Source/FlowCytometrySample are grouped by contact and date. Hence, Target/Experiment is essentially a full outer join of Source/Experiment and Source/FlowCytometrySample on contact and date. □

Manipulating Atomic Values In data integration and schema evolution, information stored as one atomic element in one database may be modeled as more than one atomic element in another. The reverse situation where multiple atomic elements in one database are combined into a single element in another database also occurs frequently in practice. Due to semantic heterogeneity, sometimes, there is also a need to apply a function on an atomic value in order to bring it into the correct semantic context in another database. A simple example would be the need to translate US dollars into Euros. In all these cases, mapping from one element to one or more elements and vice versa will typically require the application of one or more functions.

The source and target schema are shown in Fig. 2(k). The transformation assumes the existence of three functions `getFirstName()`, `getLastName()` and `concat()`. For every Source/Contact record, a Target/Contact record is created where its FirstName and LastName values are obtained by applying `getFirstName()` and `getLastName()`, respectively, on the name value of the Source/Contact record. A reverse situation occurs with Target/Contact/Address, where the Address value is the result of applying the `concat()` function on street, city and zip of Address in the Source/Contact record. □

4. COMPLEX MAPPING SCENARIO AND SOURCE INSTANCE GENERATION

In order to allow a thorough evaluation of the mapping systems, it is important for a mapping benchmark to provide scenarios and instances of different complexity and sizes. STBenchmark provides two modules for this purpose, namely SGen and IGen. SGen takes as input a set of configuration parameters and returns as output a mapping scenario $(\mathbf{S}, \mathbf{T}, \mathcal{P})$. Using appropriate configuration parameters, the mapping scenario generated by SGen can be significantly more complex and larger than the basic mapping scenarios of Sec. 3. SGen can also be used as a schema generator since one can choose to ignore $\mathbf{T}$ and $\mathcal{P}$. IGen takes as input a schema and a set of configuration parameters and returns as output an instance that conforms to the input schema. The size of the generated instance varies according to the configuration parameters.

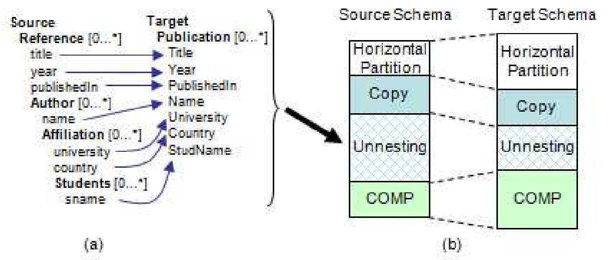


Figure 3: (a) Example of an unnesting scenario generated by SGen and (b) a concatenation of mapping scenarios by SGen.

The data model used internally by SGen and IGen is the nested relational model, which is sufficiently general to model relational schemas and a large class of XML schemas. In this model, a schema is a set of *roots* (or labels), where each root has an associated type τ. A type τ can be an atomic type (e.g., String or Int), or a Set of τ, or a record Rcd$[l_1 : \tau_1, ..., l_n : \tau_n]$, where l_i are labels and τ_i are types. In the case when each root is a Set of Rcd$[l_1 : \tau_1, ..., l_n : \tau_n]$ where τ_i are atomic types, this models a relational schema.

4.1 SGen: A mapping Scenario Generator

In what follows, we describe how SGen (i) allows a benchmark user to tune the characteristics of a generated mapping scenario through a set of configuration parameters, (ii) constructs the source and target schemas, including source and target constraints, of a complex mapping scenario from basic mapping scenarios, (iii) generates the specification of the transformation in a generated mapping scenario, (iv) ensures that generated mapping scenarios are reproducible over time, across different hardware platforms and operating systems.

(i) Input configuration parameters to SGen. SGen allows a benchmark user to tune the characteristics of generated mapping scenarios through a set of configuration parameters. These parameters essentially specify the characteristics of the schemas in the mapping scenarios to be generated. The configuration parameters consist of 6 characteristic parameters $\mathcal{C}$, along with 6 standard deviation parameters $\mathcal{D}$, and 12 repetition parameters $\mathcal{R}$ (one for each basic scenario and one for a composed scenario which will be explained shortly). The parameters in $\mathcal{C}$ are described below.

- Nesting Depth (N_d): Determines the nesting level of the schemas. If set to 0, the generated schemas are relational.
- Number of Subelements (N_s): Specifies the number of children elements of an element. In the case of relational schemas, it specifies the number of attributes in the tables.
- Join Size (N_j): Specifies the length of the join paths in the schemas.
- Join Kind (N_k): Determines the kind of the joins (star or chain) in the schemas.
- Join Width (N_w): Specifies the number of atomic subelements that participate in a join.
- Number of Function Arguments (N_a): Determines the number of source atomic elements on which a function should be applied.

Every characteristic parameter in $\mathcal{C}$ is in fact the mean value of a Gaussian distribution whose standard deviation is the value of the corresponding parameter in $\mathcal{D}$. By sampling values from these Gaussian distributions, SGen is able to generate natural looking schemas, i.e., schemas that are not rigid. For example, if N_s has a value of 3 and the corresponding standard deviation is 1, it essentially means that most complex schema elements that are generated will have between 2 and 4 subelements, and not always 3. If the standard deviation is 0, then every complex schema element will have exactly 3 subelements.

(ii) Generating Complex Mapping Scenarios. SGen generates and outputs a complex mapping scenario by concatenating multiple mapping scenarios. Intuitively, the concatenation of two scenarios $\mathcal{M}_1 = (\mathbf{S_1}, \mathbf{T_1}, \mathcal{P}_1)$ and $\mathcal{M}_2 = (\mathbf{S_2}, \mathbf{T_2}, \mathcal{P}_2)$, denoted as $\mathcal{M}_1.\mathcal{M}_2$, is a new mapping scenario $(\mathbf{S_1}.\mathbf{S_2}, \mathbf{T_1}.\mathbf{T_2}, \mathcal{P}_1.\mathcal{P}_2)$, where $\mathbf{S_1}.\mathbf{S_2}$ (resp. $\mathbf{T_1}.\mathbf{T_2}$) denotes a schema that is obtained by concatenating the roots of $\mathbf{S_1}$ (resp. $\mathbf{T_1}$) with the roots of $\mathbf{S_2}$ (resp. $\mathbf{T_2}$) and $\mathcal{P}_1.\mathcal{P}_2$ has the following semantics: Given a source instance I of $\mathbf{S_1}.\mathbf{S_2}$, the target instance of $\mathbf{T_1}.\mathbf{T_2}$ is obtained by concatenating the results of executing $\mathcal{P}_1$ and $\mathcal{P}_2$ on the parts of I that correspond to $\mathbf{S_1}$ and $\mathbf{S_2}$ respectively. Here, we assume that the symbols in $\mathbf{S_1}$ and $\mathbf{S_2}$ are disjoint and the symbols in $\mathbf{T_1}$ and $\mathbf{T_2}$ are disjoint.

The scenarios used in the concatenation by SGen are of two kinds. The first kind is referred to as *expanded* scenarios. An *expanded* scenario $\mathcal{M}^B$ is generated from one of the basic mapping scenarios described in Sec. 3, say B, using the configuration parameters $\mathcal{C}$ and $\mathcal{D}$. The transformation function $\mathcal{P}^B$ of the expanded scenario $\mathcal{M}^B{=}(\mathbf{S^B}, \mathbf{T^B}, \mathcal{P}^\mathbf{B})$ is the same as the one described by the basic mapping scenario B, except that $\mathcal{P}^B$ is now written according to the source and target schemas $\mathbf{S^B}$ and $\mathbf{T^B}$, respectively. As an example of an expanded scenario, consider the one illustrated in Fig. 3(a) which has been generated by the *Unnesting* basic mapping scenario. Observe that the specific expanded scenario has two additional levels of nesting (i.e., Affiliation and Student[4] and additional subelements when compared to the *basic unnesting scenario* of section 3. The transformation function of the expanded scenario of Fig. 3(a) can be found in Appendix B.

The second kind of scenario used by SGen in the generation of its complex mapping scenario is an intermix of different types of basic mapping scenario transformations, in which, unlike the previous kind, there may be no clean separation between a *Nesting* and a *Vertical partitioning*, for instance. This kind of scenario is referred to as *composed*. Composed scenarios intend to capture cases where different types of transformations occur simultaneously in the *same* part of the schema. They are powerful enough to describe intermixed basic transformations such as those described by GLAV mappings [26] or explicit mappings involving outer joins [34].

A composed scenario is constructed through a series of consecutive steps that are described next and illustrated through an example in Fig. 4. The role of each step it to model some of the basic mapping scenario transformations. The first step in the creation of a composed scenario is the generation of a schema $\mathbf{S}$ according to the configuration parameters $\mathcal{C}$ and $\mathcal{D}$. In the general case this will be a nested schema with referential constraints involving multiple elements. If $N_d{=}1$, the schema $\mathbf{S}$ will be relational, and if N_j or N_w are 0, then there will be no referential constraints in it. Fig. 4(a) illustrates a possible schema $\mathbf{S}$ generated in this first step. In the next step, for every root i in $\mathbf{S}$, a list F_i is constructed to contain all the atomic elements in the root i, at any depth. See Fig. 4(b). This step is performed in order to implement the *Flattening* transformation. In the third step, one of the lists F_i is randomly selected and duplicated. The new list F' is added in the list of F_is. The role of the duplicated list is to model the self join basic scenario transformation. Due to space limitations we do not perform this step in Fig. 4. To model the case in which some elements of the source schema do not appear in the target, an average of 5% of the atomic elements of the F_i lists are eliminated. Fig. 4(c) depicts the result of eliminating elements Attr$_2$, Attr$_5$ and Attr$_{10}$ from Fig. 4(b). Next, some shuffling of elements occurs across the F_i lists. In particular, two randomly selected elements from each list F_k are moved to F_{k+1} and one element from F_{k+1} to F_k. During the shuffling, the system may randomly decide to perform an element duplication instead of a move. In that case, a referential constraint is added between the two duplicates. To better understand these two tasks, Fig. 4(d) illustrates an element transfer (Attr$_{13}$) and an element duplication (Attr$_8$). The shuffling is an important task since it guarantees that the root elements in the target will not have the same atomic elements as in the source. This actually models the *denormalization* and the *vertical partitioning* transformations. A sixth step is to introduce a number of new atomic elements in some randomly selected lists F_i. In each selected list, the number of new elements is obtained by rounding up to the nearest integer 5% of the number of atomic elements that already exist in the list. This step is shown in Fig. 4(e) where the introduced elements are the Attr$_{17}$, Attr$_{18}$ and Attr$_{19}$. Each of the new elements is set either to some constant value (e.g., Attr$_{17}$), simulating that way a *Constant Value Generation* scenario, or some identifier (e.g., Attr$_{19}$), simulating the *Surrogate Key* scenario, or a value that is a function of other atomic elements (e.g., Attr$_{18}$), modeling the *Atomic Value Manipulation* scenario. The number of arguments of the functions in the latter case is also a random value that depends on the parameter N_a and its deviation value. In the sequel, the elements of each flat list F_i are nested to form a tree structure, implementing that way the *Nesting* scenario. The level of nesting and the number of atomic subelements of each element are random numbers whose value depend on the parameters N_d and N_s, respectively, and their deviation values. The root element of each such tree becomes one of the roots of a new schema $\mathbf{T}$. For our running example, the schema $\mathbf{T}$ is illustrated in Fig. 4(f). There are two additional steps that have not been mentioned here, and they serve to model the transformations of *Object Fusion* and *Horizontal Partitioning*.

It is important to note that the execution of each of the above steps is conditional on the repetition parameters $\mathcal{R}$. For instance, if the repetition parameter for the *Nesting* basic scenario is 0, then the respective step above will not be performed, ensuring that way that the target schema will not be a nested schema.

The generated composed scenario has as a source schema the schema $\mathbf{S}$ that was created in the first step (e.g., Fig. 4(a)), and as a target schema the schema $\mathbf{T}$ that was created during the last step (e.g., Fig. 4(f)). The transformation specification that we generate is a XQuery and follows the semantics of the chase with GLAV mappings, which are implicitly generated during the composed scenario generation steps. The query generation algorithm we use is similar in spirit to the query generation algorithm of [33]. We omit the discussion of this algorithm here. The transformation specification for the composed scenario example of Fig. 4 can be found in Appendix C.

To generate the final mapping scenario, SGen produces and concatenates a set of extended and composed scenarios. The mapping scenario generation algorithm steps are illustrated in Algorithm 1. SGen starts with an empty mapping scenario, i.e., a scenario with an empty source and target schema and an empty transformation function (line 1 of Algorithm 1). It then iterates through each type B of basic mapping scenario (i.e., *Copy*, *Constant Value Generation* etc.) and generates r_B extended scenarios of type B, where r_B is the value in the repetition parameter list $\mathcal{R}$ for the scenario of type B (lines 2-6 of Algorithm 1). By setting the configuration parameter r_B to 0, the benchmark user can prevent SGen from generating scenarios of type B. This is useful if the user already knows that the mapping system under evaluation does not support the respective transformation type. The mapping scenario generated so far is intended to model real life mappings which typically involve multiple types of transformations that occur in parallel and in different parts of the schemas. In the sequel, SGen generates r_S composed

[4]In general, element names may not be as meaningful as what is shown in Fig. 3(a) since we sample element names from a dictionary of English words.

Algorithm 1: SGen

```
Input:  The characteristic parameters $\mathcal{C}$,
        the corresponding standard deviations $\mathcal{D}$,
        and the list of repetition parameters $\mathcal{R}$.
Output: A mapping scenario $(\mathbf{S}, \mathbf{T}, \mathcal{P})$

SGEN($\mathcal{C}, \mathcal{D}, \mathcal{R}$)
(1)  $\mathcal{M} \leftarrow (\emptyset, \emptyset, \emptyset)$
(2)  foreach type of basic scenario $B$
(3)      Let $r_B$ be the corresponding value for scenario $B$ in $\mathcal{R}$.
(4)      foreach $i \in [1..r_B]$
(5)          $\mathcal{M}_B \leftarrow$ generateExtendedScenario($B, \mathcal{C}, \mathcal{D}$)
(6)          $\mathcal{M} \leftarrow \mathcal{M} \,.\, \mathcal{M}_B$
(7)  foreach $i \in [1..r_S]$
(8)      $\mathcal{M}_S \leftarrow$ generateComposedScenario($\mathcal{R}, \mathcal{C}, \mathcal{D}$)
(9)      $\mathcal{M} \leftarrow \mathcal{M} \,.\, \mathcal{M}_S$
(10) return $\mathcal{M}$
```

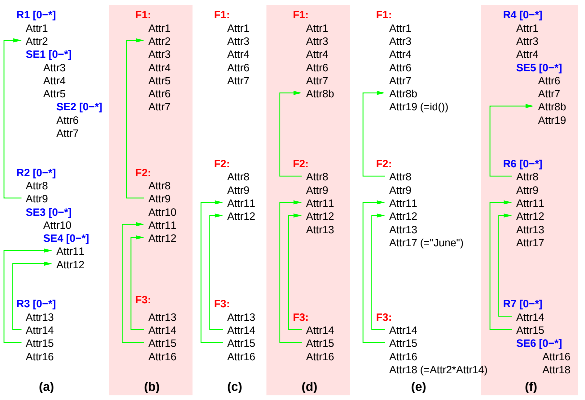


Figure 4: Composed scenario generation steps

scenarios and concatenates them to the mapping scenario that will be returned (lines 7-9 of Algorithm 1). The value r_S is the one in the repetition parameter list $\mathcal{R}$ that corresponds to the composed scenarios. By setting this parameter to 0, a benchmark user can also disallow the existence of composed scenario transformations in the generated mapping scenario. The addition of the composed scenarios in the mapping scenario generated by SGen models real life situations in which different kinds of transformations occur simultaneously at the same part of the schema. Fig. 3(b) illustrates a mapping scenario that has been constructed through this procedure. Each pair of squares with the same pattern in the source and the target schema corresponds to a component scenario. For instance, the *Horizontal Partition* type extended scenario was generated first, and it was followed by a *Copy* and an *Unnesting* type extended scenarios. Finally a composed scenario was generated and concatenated to the existing scenario.

(iii) Communicating the intended transformations.

The generated transformation function $\mathcal{P}$ is potentially complex and difficult to understand. To help the benchmark user understand the specification $\mathcal{P}$, one approach is to load the source and target schemas of the mapping scenario returned by SGen into a mapping system that supports automatic matching of source and target elements. Since our generated mapping scenario has identical source and target element names, this makes it easy for the matching module to determine the correspondences between source and target schema elements. The visual representation of the correspondences between source and target elements is thus an intuitive description of $\mathcal{P}$. However, as described in Sec. 2, the visual representation is not a reliable mechanism for describing $\mathcal{P}$ since there is currently no standard approach for interpreting such visual representations. To alleviate this problem, SGen provides hints about the transformation functions of each component mapping scenario $\mathcal{M}_1, ..., \mathcal{M}_n$ through the naming of the schema elements. For example, SGen may construct a source schema element with name "Date_2CPR2K2" in the process of generating a mapping scenario. The word "Date" is the element name. The first occurrence of the number 2 indicates that this schema element belongs to the second mapping scenario that was concatenated by SGen. Pictorially, this means that the schema element is located in the second square box from the top of the source schema in Fig. 3(b). The two letters "CP" indicate that the second square box is a *Copy* scenario. The two letters "R2" means that this schema element is a subelement of the second root in the *Copy* scenario. The letters "K2" indicate that "Date_2CPR2K2" participates in a join that involves at least two elements and "Date_2CPR2K2" is the second element in the join. It is always possible for SGen to generate such meaningful codes due to the way it constructs a mapping scenario. It is also possible to construct explanations of the transformation functions of each component scenario $\mathcal{M}_i$, $1 \leq i \leq n$, in order to explain the transformation function $\mathcal{P}$.

(iv) Determinism. An important requirement for benchmark modules that generate test cases is that, given the same input parameters, these modules must be able to reproduce the same test cases regardless of where and when they are executed. To obtain this property and to guarantee that the benchmark will easily run on different platforms, we implemented SGen in Java. The random number generators provided by the Java library, with identical seeds, produce identical streams of pseudo-random numbers that are independent of hardware platforms, operating systems, and time. Hence, given the same configuration parameters, SGen can reproduce identical mapping scenarios regardless of where and when it is executed. IGen, which we describe next, also enjoys the same determinism property.

4.2 IGen: a source Instance Generator

In this section we describe IGen, the component of STBenchmark that is used to generate instances that conform to the source schema of a mapping scenario generated through SGen. Such source instances are needed in order to perform the data transformation specified in a mapping scenario generated by SGen.

Before describing IGen, we briefly describe the ToXGene [2] data generation engine on which IGen is built upon. ToXGene takes as input a template, which describes the structure of the XML document to be generated, in a format similar to an XML schema. The input template contains annotations that specify the vocabularies and data ranges used for generating random data values, as well as the distributions used when sampling for these random values. ToXGene has a mechanism of reusing the same generated data values for different elements, providing the user with a way to enforce referential integrity constraints in the generated instance.

IGen takes as input a schema (with constraints), as well as three configuration parameters (and their associated standard deviations) which we describe next:

- N_o sets the number of occurrences of a complex element. In the nested relational model (see Sec. 4), this corresponds to the cardinality of a set of records. In the relational model, this would correspond to the number of tuples in a relation. This parameter provides the most direct way of controlling the size of the generated instance.
- S_{max} sets the maximum length of the atomic string values. By default, for string-typed atomic elements, IGen uses XMark-compliant [39] random text values, as they are generated by ToXGene. The upper bound on the length of these text values is set through S_{max}.
- N_{max} sets the upper limit for numeric values. By default, for numeric-typed atomic elements, IGen generates uniformly dis-

tributed integer values in the range from 0 to N_{max}.

Each of the above parameter has an associated standard deviation, denoted as σ_o, σ_s, and σ_n respectively. Like SGen, IGen samples values from a Gaussian distribution with mean (e.g., N_o) and standard deviation (e.g., σ_o).

Source instance generation takes place in two phases. First, IGen considers the configuration parameters described above, together with the source schema and its constraints, and outputs a ToXGene template. In the second phase, this template is provided as input to ToXGene, which in turn generates an XML instance. This process is completely transparent to the benchmark user, who only observes the final product, namely the XML instance conforming to the source schema.

5. A SIMPLE USABILITY MODEL

In this section, we describe our *simple usability (SU) model*, which is a model intended to provide a first-cut measure on the amount of effort required by a mapping designer to implement a mapping scenario through the visual interface of a mapping system. Our SU model quantifies effort roughly as the number of mouse actions and the number of keystrokes used for text input. Three different types of mouse actions are captured by our model: dragging actions, single and double mouse clicks. One might argue that there is little need to measure the exact number (e.g., x vs. $3x$) of mouse actions or keystrokes used since the task of executing the generated transformation takes a long time in general. However, since the goal of visual interfaces is to reduce a mapping designer's effort towards the generation of the desired transformation, we would like to quantify the amount of effort used in our model.

DEFINITION 5.1. The *effort* required by a mapping designer to implement a mapping scenario on a mapping system according to the SU model is a quadruple (L, S, D, K), where L is the number of dragging actions, S and D are the numbers of single mouse clicks and double mouse clicks respectively, and K is the total number of keystrokes used for text input.

For example, consider the following sequence of actions:
Right mouse click to pull up menu.
Click to select "Insert Input". Box 1 will appear on screen.
Enter name of variable "var1" in box 1.
Click to select "Specify Value".
Enter string "Testing".
Draw line from box 1 to a target schema element.

The effort required by a mapping designer in the sequence of actions above is $(1, 3, 0, 11)$ because the sequence of steps consists of 1 dragging action, 3 single mouse clicks and 11 keystrokes.

Limitations of the SU model. The SU model does not consider the aesthetics of a mapping system, such as the presentation and layout of features on the visual interface, the ease of accessing frequently used features etc. It also does not take into account human errors that may occur during the process of implementing a mapping scenario. Neither does it capture the "think time" of a mapping designer in implementing a scenario nor the amount of time required to find a particular feature (e.g., button) on the visual interface. In particular, the SU model assumes that the mapping designer is completely familiar with the visual interface of the mapping system and makes no mistakes when implementing a mapping scenario. We emphasize that the SU model is a first-cut measure on the ease of use of a mapping system. A systematic human-computer interaction study with real users (which is not the subject of this paper) would be required to make a comprehensive study on the usability of mapping systems.

A Simple Cost Model. Following the findings of [28] which show that it is generally slower and more error-prone to perform a dragging task than a point-and-click task, we assign a higher cost for dragging than for a single or double mouse click in our cost model. The cost for a double mouse click is twice that of the cost for a single mouse click since it requires twice the effort. In addition, following the intuition that it is also easy to make mistakes (e.g., typos) when typing, we model the cost of a keystroke to be equal to that of the cost of a dragging task.

DEFINITION 5.2. Let (L, S, D, K) be the effort of implementing a mapping scenario on a mapping system under the SU model. The *cost* associated with this effort is $4L + S + 2D + 4K$.

For example, the cost associated with the effort consisting of 1 dragging action, 3 single mouse clicks and 11 keystrokes is 51.

Our experience shows that the SU model is able to cover all types of actions needed to implement all mapping scenarios in every mapping system that we have considered. The details are presented in Sec. 6.1.

6. EXPERIENCE AND EXPERIMENTS

To demonstrate the wide applicability of our benchmark, we describe a few potential applications with STBenchmark and showcase the use of STBenchmark [1] by applying it to evaluate four different mapping systems.

Example applications. Schema integration is the problem of producing an integrated schema from multiple input schemas. Hence, SGen could be used to produce multiple input schemas for experimenting with schema integration algorithms. SGen could also be used to produce pairs of schemas to test schema matching algorithms. Another example that requires generating mapping scenarios, in addition to schemas, to test the algorithms that were developed is that of schema evolution. Indeed, SGen could replace the custom-made schema evolution scenario generators in [48, 44, 4] that were developed specially for their experiments. As another application, in the area of debugging schema mappings, algorithms have been developed to compute routes that show the relationships of tuples in the source and target instance of a data exchange [15]. SGen and IGen, together with a schema mapping tool that is used to obtain a target instance, could be employed to create synthetic mapping scenarios for these route computation algorithms. We note that SGen's ability to generate constraints in the target schema is extremely useful for testing the route algorithms.

Next, we showcase the use of STBenchmark by applying it to evaluate four mapping systems.

6.1 Usability Experience

In this section, we report our usability experience of each mapping system. Our goal is to implement each basic mapping scenario through the visual interface of each mapping tool and report on the ease/difficulty of doing so. One method of quantifying the degree of ease/difficulty is through the cost and effort measures as described in Sec. 5.

Methodology. Prior to implementing each basic mapping scenario, we have familiarized ourselves with each mapping system and can thus be considered expert users of each mapping system. This means that when there are multiple ways of implementing a mapping scenario, we assume that we know the method of implementing the mapping scenario with the least cost and this is the cost we report in our findings. Each mapping scenario is implemented using the default visual interface of the mapping system as customizing the visual interface may decrease the cost of implementing certain mapping scenarios. Except when mentioned, we generate execution code in the XSLT 1.0 language, which is the language that is commonly supported across all four mapping systems. In our implementations, the action of drawing a line to associate a source schema element with a target schema element is classified as a dragging action and a right mouse click is classified as a single mouse click.

Scenario / Mapping System	A		B		C		D	
	Effort	Cost	Effort	Cost	Effort	Cost	Effort	Cost
Copy	(1,4,0,0)	8	(4,0,0,0)	16	(4,0,0,0)	16	(0,4,0,0)	4
Constant Value Generation	(2,4,0,17)	80	(2,0,0,17)	76	(4,0,0,17)	84	(0,6,0,17)	74
Horizontal Partition	(10,4,0,7)	72	(9,4,1,9)	78	(8,5,0,7)	65	(0,22,2,21)	110
Surrogate Key Assignment	(14,6,0,7)	90	(13,6,1,9)	96	(8,6,0,28)	150	(0,42,2,39)	202
Vertical Partition	(7,4,0,0)	32	(12,4,0,0)	52	(9,1,0,0)	37	(0,7,0,0)	7
Unnesting	(5,0,0,0)	20	(6,2,0,0)	26	(6,1,0,0)	25	(0,7,0,0)	7
Nesting	*	*	(8,11,6,54) (a)	271 (a)	*	*	(0,18,2,0)	22
Self Joins	(18,10,0,7) (b)	110 (b)	*	*	*	*	*	*
Denormalization	*	*	*	*	*	*	(0,23,2,1)	31
Keys and Object Fusion	(12,10,0,0) (c)	58 (c)	*	*	*	*	(0,30,4,0)	38
Manipulation of Atomic Values	(12,6,0,3)	66	(11,9,2,6)	81	(15,5,0,0)	65	(0,20,0,45)	200

Legend
(*) Cannot be fully implemented through the visual interface. Requires inspection and manual modification of partially generated XSLT code.
(a) Requires the use of XSLT 2.0 specific features.
Some manual modifications required on the generated XSLT code because of an apparent bug in the mapping system.
(b) Requires that the source schema be duplicated through the visual interface.
(c) Requires that part of the target schema be duplicated through the visual interface.

Table 1: Effort and cost of implementing each basic mapping scenario.

Observations. Fig. 6.1 tabulates our findings in terms of effort and cost. We first state some cost-independent observations.

(1) **A** is the only system that is able to implement the self joins scenario. This is because **A** allows one to duplicate the source schema through the visual interface and perform a join with the original schema.

(2) **C** could not implement the most number of basic mapping scenarios. In addition, for those mapping scenarios which could be implemented, some of the generated execution code contains non-standard XSLT 1.0 code that can only be executed using the evaluation engine that comes with the system.

(3) All mapping scenarios, except for self joins, can be implemented through the visual interface of **D**. Every other mapping system could not implement at least two basic mapping scenarios. Furthermore, **D** is the only system that is capable of implementing the denormalization scenario.

We state our cost-dependent observations next. Note that except for the copy and constant value generation scenarios, the costs associated with **D** are either significantly higher or lower than the rest of the systems.

(1) The costs of implementing the copy scenario in **A** and **D** are lower than those of **B** and **C** for the following reason: While **B** and **C** support only dragging actions to relate source elements to target elements[5], **A** and **D** do not. In fact, observe that the effort for dragging actions for **D** is always 0 for every mapping scenario. To relate the source element Source/Protein/name to the target element Target/Protein/Name in **D** for example, one is required to click on the former element, click on the latter element and then perform a right mouse click action to bring up a menu and another click to confirm the correspondence. This sequence of actions has a total cost of 4 units which is in fact the same as the cost of a dragging action from the former to the latter element. The lower cost of **D** is due to the schema matching module of **D** which kicks in automatically, after the first correspondence is established, to suggest matching the corresponding accession elements and created elements in the source and target schemas. The mapping designer only needs to perform an additional mouse click to confirm all suggestions (cost of 1 unit). Furthermore, in **D**, there is no need to relate the protein elements in the two schemas to generate the correct XSLT script. Hence the total cost to implement the copy scenario in **D** is 4 units. **A** also has a schema matching module but one is required to first draw a line (which involves a dragging action) between the protein elements of the two schemas before manually invoking this module (which involves 4 single mouse clicks) to relate children elements.

[5]A total of four dragging actions are required to match the corresponding protein, name, accession and created elements in the two schemas and hence the total cost of 16 units.

(2) The cost to implement the vertical partitioning scenario for **D** is much lower than the rest of the systems. There are two reasons: First, the schema matching module of D automatically kicks in and "does the right thing". That is, the lines inferred by the matching module go hand-in-hand with the semantics of the intended transformation. Second, **D** automatically handles the target schema constraint on CoFactor in its XSLT code generation. In contrast, **A**, **B** and **C** require the designer to explicitly correlate the two target relations by CoFactor in order to achieve the desired effect.

It also costs less to implement the unnesting scenario with **D** when compared to the rest. Again, this is because **D** is the only system in which the schema matching module can be used effectively to achieve the desired visual specification.

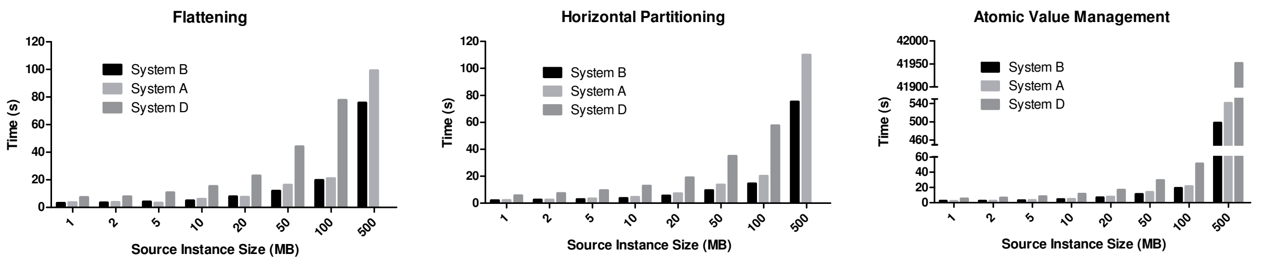
(3) For implementing surrogate key assignment and manipulation of atomic values scenarios, **D** is costlier than **A**, **B**, and **C** by more than 50 units. This is because in **D**, the mapping designer is required to enter the names of the XSLT functions used (e.g., "substring-before" to extract the first name). In contrast, one can drag the desired function from a library of supported functions in **A**, **B**, and **C**. Since our cost model heavily penalizes keyboard-based input, the cost for **D**'s implementation is high in these cases.

6.2 Performance Comparisons

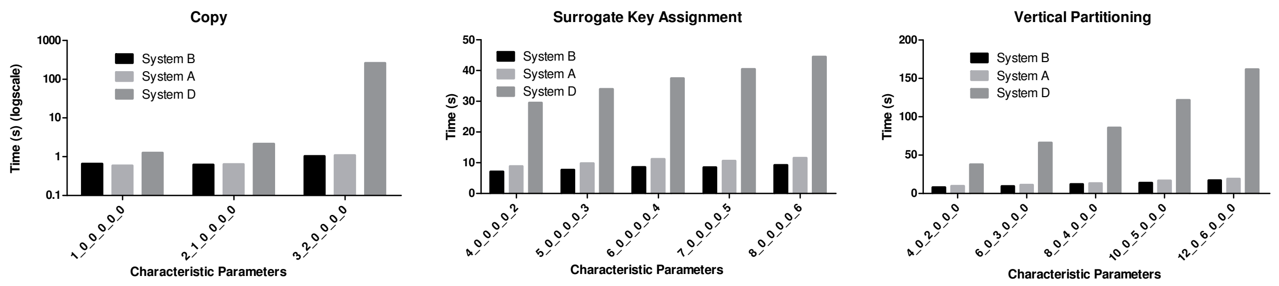
In this section we report on our experiments comparing the performance of the transformation scripts generated by the four mapping systems. Although different mapping systems could support different transformation languages, XSLT 1.0 is the only language supported by all of them. Thus, it was the natural choice for achieving meaningful comparisons between the systems. As a consequence, XML was also the data type used for the experiments.

We conducted two series of experiments. The first one determines how the size of the source instance influences the time needed to perform the data transformation through the XSLT scripts generated by the systems. The second series of experiments investigates the influence of the size of the mapping scenario on the time needed to perform the data transformation.

Experimental Setting. All of our experiments were performed on a Dual Intel Xeon 3.4 GHz workstation with 4GB RAM running GNU/Linux. To have meaningful and comparable results, we used a



(a) Source instance scaling experiments



(b) Mapping scenario scaling experiments

Figure 5: XSLT Scripts Performance Comparisons

common evaluation engine (Saxon-B version 8.9 [38]) to execute the XSLT scripts generated by all the mapping systems. Since system $\mathbf{C}$ generates non-standard XSLT code that can only be executed using the engine built in their system, we excluded system $\mathbf{C}$ from our performance comparisons.

Methodology. First, we restricted our attention, among the mapping scenarios presented in section 3, to the ones that could be implemented entirely through the graphical interface by at least two of the mapping systems $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$. One exception is the nesting scenario, which can be implemented by both $\mathbf{B}$ and $\mathbf{D}$, but was nevertheless discarded from our experimental comparison. The reason is that $\mathbf{B}$ can generate XSLT 2.0 code that takes advantage of grouping constructs built into the language, whereas $\mathbf{D}$ has to achieve the same effect through XSLT 1.0 constructs. This leads to uncomparable experimental results.

Source Instance Scaling Experiments. In the first set of experiments, for each basic scenario, we keep the source and target schemas fixed (as they are presented in Sec. 3). Using IGen, we generated multiple source instances of varying sizes.

In Fig. 5(a), we show only the results we obtained for the flattening, horizontal partitioning and atomic value management basic scenarios. The results for the other scenarios show a similar trend and are thus omitted. The size of the source instance (in MB) is represented on the X axis, while the execution time (in seconds) is represented on the Y axis.

For source instances of size no greater than 100MB, all the processing could be done in main memory. However, for larger instances, (e.g., 500MB) we noticed significant disk swapping during the execution of the XSLT scripts. Hence, the observed execution times for 500MB instances are much longer than the times for smaller instances. For the flattening and horizontal partitioning scenarios, the scripts generated by $\mathbf{D}$ could not be executed on 500MB instances due to insufficient memory.

As a general observation, the scripts generated by $\mathbf{B}$ are the fastest to execute, followed by $\mathbf{A}$, and the transformation scripts generated by $\mathbf{D}$ are the slowest. Throughout the experiments, the gap between $\mathbf{B}$ and $\mathbf{A}$ was much less significant than the one between $\mathbf{A}$ and $\mathbf{D}$. By inspecting the XSLT scripts generated by the systems, we can identify some possible causes for the differences in execution times, which we detail below:

First, one peculiarity of the scripts generated by $\mathbf{A}$ is that they employ `for-each` loops with `value-of` statements nested inside, in order to access the value of an atomic element in any context, even if the cardinality of the atomic element is at most one. On the other hand, the scripts generated by $\mathbf{B}$ use directly `value-of` statements relative to the current context to achieve the same goal. Since atomic elements with a cardinality of one represent a very frequent case in our scenarios, the redundant `for-each` loops account for a significant part of the difference in execution time between scripts generated by $\mathbf{B}$ and $\mathbf{A}$.

Another cause of overhead in the scripts of $\mathbf{A}$ is the way they make use of variables. For instance, to make a string comparison, a script of $\mathbf{A}$ would first store the Boolean result of the comparison in a variable, then convert the value of this variable to a string, and finally make a second comparison between this value and the literals `true` or `false`. Also, to produce a value through a sequence of function applications, the scripts of $\mathbf{A}$ store each intermediate result in a separate variable and use this variable as an argument to subsequent function applications. In contrast, the scripts of $\mathbf{B}$ employ directly an expression containing the composed function applications, without storing intermediate results in temporary variables.

The main cause of the observed differences in execution time between the scripts generated by $\mathbf{D}$ and the others lies in the fact that $\mathbf{D}$ performs the data transformation through a two-phased process, where each phase involves the execution of a separate XSLT script. Also, the script executed in the first phase assigns an identifier to each element of a complex type generated in the intermediate instance. Computing the value of this identifier is potentially expensive since it involves accessing the values of all the atomic elements in the source instance that contribute to the complex element being generated.

Mapping Scenario Scaling Experiments. In the second series of experiments, we investigate the influence of the size of the mapping scenario on the time needed to execute the XSLT scripts generated by the mapping systems. We performed one group of experiments for each mapping scenario described in Sec. 3. The results for the copy, surrogate key assignment, and vertical partitioning scenarios are presented in Fig. 5(b). The graphs we obtained for the other basic scenarios exhibited a similar trend and we omit them here.

As presented in Sec. 4.1, SGen produces a mapping scenario based on configuration parameters $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{R}$. The mean values used for each of the 6 characteristic parameters in $\mathcal{C}$ are represented on the X axis of the graphs (in the vector $N_d\_N_s\_N_j\_N_k\_N_w\_N_a$), while the execution time of the scripts (in seconds) is represented on the Y axis. For the join kind N_k parameter we used a value of 0 for star joins and 1 for chain joins. To ensure that we are able to sample larger values as we increase the vector of mean values, we set the standard deviations of every parameter in $\mathcal{D}$ to 0. Except for the basic scenario of interest for which the repetition parameter is set to 1, every other repetition parameter in $\mathcal{R}$ is set to 0. We started with a set of initial mean values for $\mathcal{C}$ To achieve the desired scenario scaling effect, we increased simultaneously the mean values of all relevant characteristic parameters, thus obtaining, at each step, a larger mapping scenario.

To make the comparisons meaningful, in each experiment, we kept the parameters used for generating the synthetic source instance using IGen (see Sec. 4.2) constant across the different schema configurations. We then implemented each scenario through the graphical interfaces of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{D}$. As in the previous series of experiments, we compared the execution times of the XSLT scripts generated by each system.

There are some interesting observations we can make by analyzing the results of this second set of experiments. For the copy scenario, the scripts generated by $\mathbf{D}$ are orders of magnitude slower to execute than the scripts generated by the two other systems (observe the logarithmic scale of the time axis in the first graph of Fig. 5(b)). This behavior appears if the source and target schemas (identical in the case of the copy scenario) are nested. The XSLT script generated by $\mathbf{D}$ contains a nested `for-each` loop with as many levels as the levels of nesting in the source schema. The domain of each `for-each` iteration is a nodeset returned by an XPath expression relative to the root of the source instance. Evaluating these XPath expressions at each iteration in the nested loop becomes very expensive on large source schemas and instances. This accounts for the blowup in execution times of $\mathbf{D}$ scripts when introducing nesting in the source schemas used for the copying scenario.

In the surrogate key assignment and vertical partitioning scenarios, there is an additional reason for the differences observed in execution times. While the scripts of $\mathbf{A}$ and $\mathbf{B}$ use the built-in `generate-id` XSLT function to construct identifiers, $\mathbf{D}$'s method of constructing identifiers does not rely on any XSLT built-in functions or any language-specific constructs for that matter. The `generate-id` function is a function of the first node in the node-set given as an argument, while $\mathbf{D}$'s identifier-generating function constructs a string from a sequence of atomic-valued expressions. This method of constructing identifiers is more expensive since $\mathbf{D}$ is required to obtain all values of the atomic-valued expressions before an identifier can be constructed.

7. RELATED WORK

Benchmarks have been developed for many different areas, such as data mining, dependable systems and data stream systems. Perhaps the most notable benchmarks are the ones for relational or XML query engines [8, 12, 37, 39, 42, 46]. Benchmarks for query engines typically consist of a set of queries on a predefined schema, as well as a data generation module that produces instances of different sizes according to the schema. The queries are designed to test various features that are typically expected to be supported by a query engine. The benchmark can be used to determine the extent of queries supported by a query engine, as well as how well the performance of the query engine scales with the size of the input instances. STBenchmark is similar to such benchmarks. The basic mapping scenarios and the generators, SGen and IGen, serve a similar purpose.

Benchmarks have also been developed for schema matching systems. In XBenchMatch [17], a matching system is evaluated by measuring the precision and recall (similar to the way they are used in information retrieval) of the results returned by the matching system with respect to the correct matchings. In STBenchmark, one aspect of the evaluation of a mapping system is to test whether each basic mapping scenario can be implemented through its visual interface. It does not measure the precision and recall, or the extent to which an implementation of a mapping scenario can be achieved. Another benchmark for ontology/schema matching is OAEI [31], which aims at building evaluation methods and collecting test cases for ontology/schema matchings. STBenchmark is not about collecting test cases although it would be useful to have a repository of mapping scenarios, especially when a standard for specifying mapping scenarios as input to mapping systems exists.

In [25], the authors gave a detailed study on the different kinds of matchings that can exist between schema elements. Their findings can serve as a guide on some of the basic transformations that should be directly supported by a mapping system. Some of our basic mapping scenarios coincide with their findings.

One approach in measuring data exchange systems is to evaluate the process from end-to-end, i.e., considering the matching, the mapping and the query execution time as one single process and evaluate the final result. Based on this idea, Spicy [10] uses stractural analysis to compare a target instance to another one used as a reference, and deciding that way the quality of the mapping. For benchmarking ETL transformations, on the other hand, factors like complexity and execution time are of major importance [43].

In [32], a framework for developing benchmarks for data exchange tools has been proposed. However, no actual benchmark has been implemented. The proposed schema generation algorithm in [32] follows a top down approach, where parts of a large schema are projected to create different mapping scenarios. In contrast, our scenario generation strategy is bottom up. It scales and combines basic mapping scenarios in different ways to form larger complex mapping scenarios. For this reason, we believe the space of possible mapping scenarios that we can generate is much larger.

THALIA [22] is a benchmark for information integration systems. It describes a set of benchmark queries and the respective schemas these queries are based upon. The integration system is evaluated by counting the number of queries that return correct results and the effort required to reconcile the schemas. In contrast to STBenchmark, it does not provide synthetic schemas, thus it is hard to evaluate how well the integration systems scale. Furthemore, it does not provide a detailed model for measuring the required integration effort.

Various schema and evolution scenario generators have also been proposed in different research projects for their experimental settings [4, 47, 48]. However, to the best of our knowledge, SGen is the first general-purpose *mapping scenario* generator.

Visual interfaces for specifying XML-to-XML [11, 18] or relational-to-relational [49] transformations, in XQuery or XML, respectively, can also be considered as mapping systems. Hence, it is also possible to apply STBenchmark on these systems to evaluate the expressiveness, performance of queries generated, and the usability of these systems.

8. CONCLUSIONS

We have described STBenchmark, a benchmark that we have developed for evaluating mapping systems. STBenchmark consists of three components that are designed to test how readily mapping systems support basic mapping scenarios, how well they perform (in this paper, in terms of the performance of generated XSLT scripts) on simple and complex mapping scenarios with instances of varying sizes, as well as the ease of use of mapping systems. We have evaluated four mapping systems with STBenchmark. In addition to our findings reported in Sec. 6, our experience with mapping systems indicate that mapping systems share a lot of commonality in terms of the visual metaphors and constructs used for designing mappings. However, there is currently a lack of standard in interpreting these visual metaphors and constructs. Hence, even though mapping systems take visual specifications as input, the transformation functions in our basic mapping scenarios are specified with XQuery and we leave the task of creating the visual specifications of mapping scenarios on mapping systems to the benchmark user.

Based on our study, we believe that it is crucial to develop a standard (either by standardizing the interpretation of visual metaphors and constructs and extending upon them to achieve more expressiveness (e.g., [34]) or, by developing a standard mapping specification language) for specifying inputs to mapping systems. Such a standard will not only serve to standardize the specifications of basic mapping scenarios in STBenchmark and output of SGen, but also serve as an important step towards the development of a uniform testbed and repository for schema mappings and data exchange tasks [6].

Acknowledgment: Work partly supported by the NSF CAREER Award IIS-0347065, the NSF grant IIS-0430994 and the EU Marrie-Currie Fellowship MIRG-CT-2006-046548.

References

[1] B. Alexe, W. Tan, and Y. Velegrakis. Comparing and Evaluating Mapping Systems with STBenchmark. In *VLDB*, 2008.

[2] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons. Toxgene: An extensible template-based data generator for xml. In *WebDB*, pages 49–54, 2002.

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

[4] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash. Implementing Mapping Composition. In *VLDB*, pages 55–66, 2006.

[5] P. A. Bernstein, S. Melnik, and J. E. Churchill. Incremental Schema Matching. In *VLDB (demo)*, pages 1167–1170, 2006.

[6] Bertinoro Workshop on Information Integration. http://www.dis.uniroma1.it/~lenzerin/INFINT2007/.

[7] Biowarehouse database integration for bioinformatics. http://biowarehouse.ai.sri.com/.

[8] T. Bohme and E. Rahm. XMach-1: A Benchmark for XML Data Management. In *BTW*, pages 264–273, 2001.

[9] A. Bonifati, E. Q. Chang, T. Ho, and L. V. S. Lakshmanan. HepToX: Heterogeneous Peer to Peer XML Databases, 2005.

[10] A. Bonifati, G. Mecca, A. Pappalardo, S. Raunich, and G. Summa. Schema mapping verification: the spicy way. In *EDBT*, pages 85–96, 2008.

[11] D. Braga, A. Campi, and S. Ceri. *QBE* (*q*uery *b*y *e*xample): A visual interface to the standard xml query language. *ACM TODS*, 30(2):398–443, 2005.

[12] S. Bressan, G. Dobbie, Z. Lacroix, M.-L. Lee, Y. G. Li, U. Nambiar, and B. Wadhwa. X007: Applying 007 Benchmark to XML Query Processing Tool. In *CIKM*, pages 167–174, 2001.

[13] M. J. Carey. Data delivery in a service-oriented world: the BEA AquaLogic data services platform. In *SIGMOD Conference*, pages 695–705, 2006.

[14] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. Xperanto: Middleware for publishing object-relational data as xml documents. In *VLDB*, pages 646–648, 2000.

[15] L. Chiticariu and W. Tan. Debugging Schema Mappings with Routes. In *VLDB*, pages 79–90, 2006.

[16] dblp.uni-trier.de: Computer science bibliography. http://dblp.uni-trier.de/.

[17] F. Duchateau, Z. Bellahsene, and E. Hunt. XBenchMatch: a Benchmark for XML Schema Matching Tools. In *VLDB*, pages 1318–1321, 2007.

[18] M. Erwig. Xing: a visual xml query language. *J. Vis. Lang. Comput.*, 14(1):5–45, 2003.

[19] L. M. Haas. Beauty and the Beast: The Theory and Practice of Information Integration. In *ICDT*, pages 28–43, 2007.

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

[21] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation for large-scale semantic data sharing. *VLDB J.*, 14(1):68–83, 2005.

[22] J. Hammer, M. Stonebraker, and O. Topsakal. THALIA: Test Harness for the Assessment of Legacy Information Integration Approaches. In *ICDE*, pages 485–486, 2005.

[23] Rational Data Architect. www.ibm.com/software/data/integration/rda.

[24] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, pages 61–75, 2005.

[25] F. Legler and F. Naumann. A Classification of Schema Mappings and Analysis of Mapping Tools. In *BTW*, pages 449–464, 2007.

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

[27] B. S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. *TODS*, 25(1):83–127, Mar. 2000.

[28] I. S. MacKenzie, A. Sellen, and W. Buxton. A comparison of input devices in elemental pointing and dragging tasks. In *CHI*, pages 161–166, 1991.

[29] Altova MapForce Professional Edition, Version 2008. http://www.altova.com.

[30] Merriam-Webster. Merriam-Webster Online - The Language Center. http://www.m-w.com/home.htm.

[31] http://oaei.ontologymatching.org.

[32] T. Okawara, A. Morishima, and S. Sugimoto. An Approach to the Benchmark Development for Data Exchange Tools. In *Databases and Applications*, pages 19–25, 2006.

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

[34] A. Raffio, D. Braga, S. Ceri, P. Papotti, and M. A. Hernández. Clip: a Visual Language for Explicit Schema Mappings. In *ICDE*, 2008.

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

[36] M. Roth, M. A. Hernández, P. Coulthard, L. Yan, L. Popa, H. C.-T. Ho, and C. C. Salter. XML mapping technology:Making connections in an XML-centric world. *IBM Sys. Journal*, 45(2):389–410, 2006.

[37] K. Runapongsa, J. M. Patel, H. V. Jagadish, and S. Al-Khalifa. The Michigan Benchmark: A Microbenchmark for XML Query Processing Systems. In *EEXTT*, pages 160–161, 2002.

[38] The Saxon XSLT and XQuery Processor, version 8.9. http://saxon.sourceforge.net.

[39] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985, 2002.

[40] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB J.*, 10(2-3):133–154, 2001.

[41] Stylus Studio, XML Enterprise Suite, Release 2. http://www.stylusstudio.com.

[42] TPC Transaction Processing Performance Council. http://tpc.org.

[43] P. Vassiliadis, A. Karagiannis, V. Tziovara, and A. Simitsis. Towards a Benchmark for ETL Workflows. In *QDB*, pages 49–60, 2007.

[44] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.

[45] Microsoft Visual Studio 2005, Version 8.0.50727.42. http://msdn2.microsoft.com/en-us/ie/bb188238.aspx.

[46] B. Yao, T. Ozsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *ICDE*, pages 621–633, 2004.

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

[48] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, pages 1006–1017, 2005.

[49] M. Zloof. Query-By-Example: A Data Base Language. *IBM Sys. Journal*, 16(4):324–343, 1977.

APPENDIX

A. BASIC MAPPING SCENARIOS

A.1 Copying

The copying scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(a). The precise specification of $\mathcal{P}$ is described in XQuery below:

```
<Target>
for $x0 in /Source/Protein
return
<Protein>
   <Name> $x0/name/text()</Name>
   <Accession> $x0/accession/text()</Accession>
   <Created> $x0/created/text()</Created>
</Protein>
</Target>
```

A.2 Constant Value Generation

The constant value generation scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{T}$ is shown in Fig. 2(b). The source schema can be any schema and is therefore not shown. The precise specification of $\mathcal{P}$ is described in XQuery below:

```
<Target>
<DataSet>
   <Name>SwissProt</Name>
   <LoadingDate>July 4th</LoadingDate>
</DataSet>
</Target>
```

An alternative to the constant date value "July 4th" is a system function `today()` that returns the current date. However, the use of such system functions is dependent on the mapping system.

A.3 Horizontal Paritioning

The horizontal partitioning scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(c). For this mapping scenario, we ignore the WID subelements in Target/Gene and Target/Synonym. The precise specification of $\mathcal{P}$ is described in XQuery below.

```
<Target>
for $x0 in /Source/Gene
where $x0/type/text() = 'primary'
return
  <Gene>
     <Name> $x0/name/text() </Name>
     <Protein> $x0/protein/text() </Protein>
  </Gene>
for $x0 in /Source/Gene
where $x0/type/text() != 'primary'
return
   <Synonym>
      <Name> $x0/name/text() </Name>
      <Protein> $x0/protein/text() </Protein>
   </Synonym>
</Target>
```

A.4 Surrogate Key Assignment

The surrogate key assignment scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(d) and is identical to the source and target schemas, respectively, of the horizontal partitioning scenario. Assuming that `genID()` is an XQuery function that returns a new identifier each time it is invoked, the precise specification of $\mathcal{P}$ is described in XQuery below.

```
<Target>
for $x0 in /Source/Gene
where $x0/type/text() = 'primary'
return
<Gene>
   <Name> $x0/name/text() </Name>
   <Protein> $x0/protein/text() </Protein>
   <WID> genID() </WID>
</Gene>
for $x0 in /Source/Gene
where $x0/type/text() != 'primary'
return
<Synonym>
   <Name> $x0/name/text() </Name>
   <Protein> $x0/protein/text() </Protein>
   <WID> genID() </WID>
</Synonym>
</Target>
```

A.5 Vertical Partition

The vertical partition scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(e). Assuming that `genID(.)` is an XQuery function that returns an identifier based on its input arguments each time it is invoked, the precise specification of $\mathcal{P}$ is described in XQuery below. In our XQuery specification below, we sometimes omit the end tags to save space.

```
<Target>
for $x0 in /Source/Reaction
let $id = genID($x0)
return
   <Reaction>
      <Entry> $x0/name/text()
      <Name> $x0/name/text()
      <Comment> $x0/comment/text()
      <Orthology> $x0/orthology/text()
      <CoFactor> $id </CoFactor>
   </Reaction>
for $x0 in /Source/Reaction
let $id = genID($x0)
return
   <ChemicalInfo>
      <Definition> $x0/definition/text()
      <Equation> $x0/equation/text()
      <CoFactor> $id
   </ChemicalInfo>
</Target>
```

A.6 Unnesting (or Flattening)

The unnesting scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(f). The precise specification of $\mathcal{P}$ is described in XQuery below. As before, we sometimes omit the end tags to save space.

```
<Target>
for $x0 in /Source/Reference
    $x1 in $x0/Author
return
   <Publication>
      <Title> $x0/title/text()
      <Year> $x0/text()
      <PublishedIn> $x0/publishedIn/text()
      <Name> $x1/name/text()
   </Publication>
</Target>
```

A.7 Nesting

The nesting scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(g). The precise specification of $\mathcal{P}$ is described in XQuery below.

```
<Target>
for $x0 in distinct-values(
                 /Source/Reference/year)
return
   <Period>
      <Year> $x0
      for $x1 in distinct-values(
                 /Source/Reference[year=$x0]/name)
         return
            <Author>
               <Name> $x1
               for $x2 in /Source/Reference
               where $x2/year/text()=$x0 and
                     $x2/name/text()=$x1
               return
                  <Publication>
                     <Title> $x2/title/text()
                     <PublishedIn>
                        $x2/publishedIn/text()
                     </PublishedIn>
                  </Publication>
            </Author>
      </Period>
</Target>
```

A.8 Self Joins

The self joins scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(h). The precise specification of $\mathcal{P}$ is described in XQuery below.

```
<Target>
for $x0 in /Source/Gene
where $x0/type/text() = 'primary'
return
   <Gene>
      <Name> $x0/name/text()
      <Protein> $x0/protein/text()
   </Gene>

for $x0 in /Source/Gene
    $x1 in /Source/Gene
where $x0/type/text() = 'primary' and
      $x1/type/text() != 'primary' and
      $x1/protein/text() = $x0/protein/text()
   return
      <Synonym>
         <Name> $x1/name/text()
         <WID> $x0/name/text()
      </Synonym>
</Target>
```

A.9 Denormalization and Join Path Selection

The denormalization and join path selection scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(i). The precise specification of $\mathcal{P}$ is described in XQuery below.

```
<Target>
for $x0 in $doc/Source/Name,
    $x1 in $doc/Source/Node
where $x0/id/text() = $x1/taxId/text()
return
   <Taxon>
      <Id> $x0/id/text()
      <Name> $x0/name/text()
      <UniqueName> $x0/uniqueName/text()
      <Class> $x0/class/text()
      <Parent> $x1/parentId/text()
      <Rank> $x1/rank/text()
      <EmblCode> $x1/emblCode/text()
  </Taxon>
</Target>
```

A.10 Keys and Object Fusion

The keys and object fusion scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(j). We assume that contact and date form a key for the Experiment and FlowCytometrySample sets. The precise specification of $\mathcal{P}$ is described in XQuery below. The first part creates a target Experiment element for each source Experiment and copies the associated ExperimentalData information, as well as the ExperimentalData associated with any FlowCytometrySample that agrees with the source Experiment on contact and date. The second part creates target Experiment elements for each FlowCytometrySample that does not agree on contact and date with any source Experiment.

```
<Target>
for $x0 in /Source/Experiment
return
   <Experiment>
      <Contact> $x0/contact/text()
      <Date> $x0/date/text()
      <Description> $x0/description/text()
      for $x1 in $x0/ExperimentalData
      return
         <ExperimentalData>
            <Data> $x1/data/text()
            <Role> $x1/role/text()
         </ExperimentalData>
      for $x2 in /Source/FlowCytometrySample
      where $x2/contact=$x0/contact
            and $x2/date=$x0/date
      return
         for $x3 in $x2/Probe
         return
            <ExperimentalData>
                <Data> $x3/data/text()
                <Role> $x3/type/text()
            </ExperimentalData>
   </Experiment>

for $x0 in /Source/FlowCytometrySample
where
   not(exists(/Source/Experiment
        [contact=$x0/contact and date=$x0/date]))
return
   <Experiment>
      <Contact> $x0/contact/text()
      <Date> $x0/date/text()
      <Description/>
      for $x1 in $x0/Probe
      return
          <ExperimentalData>
              <Data> $x1/data/text()
              <Role> $x1/type/text()
          </ExperimentalData>
   </Experiment>
</Target>
```

A.11 Manipulating Atomic Values

The keys and object fusion scenario is $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 2(k). The precise specification of $\mathcal{P}$ is described in XQuery below.

```
<Target>
for $x0 in $doc/Source/Contact
return
   <Contact>
      <FirstName> getFirstName($x0/name/text())
      <LastName> getLastName($x0/name/text())
      <Address> concat($x0/street/text(),
                       $x0/city/text(),
                       $x0/zip/text())
      <Phone> $x0/phone/text()
   </Contact>
</Target>
```

B. AN EXPANDED SCENARIO

The precise specification of the transformation function $\mathcal{P}$ of the scenario $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 3(a) is provided next in XQuery. Note that this scenario is an expanded scenario generated by the *Unnesting* basic scenario.

```
<Target>
for $x0 in /Source/Reference,
    $x1 in $x0/Author,
    $x2 in $x1/Affiliation,
    $x3 in $x2/Student
return
   <Publication>
      <Title> $x0/title/text()
      <Year> $x0/text()
      <PublishedIn> $x0/publishedIn/text()
      <Name> $x1/name/text()
      <University> $x2/university/text()
      <Country> $x2/country/text()
      <StudName> $x3/sname/text()
   </Publication>
</Target>
```

C. A COMPOSED SCENARIO

The precise specification of the transformation function $\mathcal{P}$ of the scenario $(\mathbf{S}, \mathbf{T}, \mathcal{P})$, where $\mathbf{S}$ and $\mathbf{T}$ are shown in Fig. 4(a) and Fig. 4(f), respectively, is provided next in XQuery.

```
<Target>
for $x0 in /Source/R1,
    $x1 in $x0/SE1, $x2 in $x1/SE2,
    $x3 in /Source/R2,
    $x4 in $x3/SE3, $x4 in $x3/SE4,
    $x5 in /Source/R3
where
   $x0/Attr2/text()=$x3/Attr9/text() and
   $x5/Attr14/text()=$x4/Attr12/text() and
   $x5/Attr15/text()=$x4/Attr11/text()
return
   <R4>
     <Attr1> $x0/Attr1/text()
     <Attr3> $x1/Attr3/text()
     <Attr4> $x1/Attr4/text()
     for $x10 in /Source/R1,
         $x11 in $x0/SE1, $x12 in $x1/SE2,
         $x13 in /Source/R2,
         $x14 in $x3/SE3, $x14 in $x3/SE4,
         $x15 in /Source/R3
     where
        $x10/Attr2/text()=$x13/Attr9/text() and
        $x15/Attr14/text()=$x14/Attr12/text() and
        $x15/Attr15/text()=$x14/Attr11/text() and
        $x10/Attr1/text()=$x0/Attr1/text() and
        $x11/Attr3/text()=$x1/Attr3/text() and
        $x11/Attr4/text()=$x1/Attr4/text()
     return
        <SE5>
          <Attr6> $x12/Attr6/text()
          <Attr7> $x12/Attr7/text()
          <Attr8b> $x13/Attr8/text()
          <Attr19> id()
        </SE5>
   </R4>

for $x0 in /Source/R1,
    $x1 in $x0/SE1, $x2 in $x1/SE2,
    $x3 in /Source/R2, $x4 in $x3/SE3,
    $x4 in $x3/SE4,
    $x5 in /Source/R3
where
    $x0/Attr2/text()=$x3/Attr9/text() and
    $x5/Attr14/text()=$x4/Attr12/text() and
    $x5/Attr15/text()=$x4/Attr11/text()
return
    <R6>
      <Attr8>  $x3/Attr8/text()
      <Attr9>  $x3/Attr9/text()
      <Attr11> $x4/Attr11/text()
      <Attr12> $x4/Attr12/text()
      <Attr13> $x5/Attr13/text()
      <Attr17> June
    </R6>

for $x0 in /Source/R1,
    $x1 in $x0/SE1, $x2 in $x1/SE2,
    $x3 in /Source/R2,
    $x4 in $x3/SE3, $x4 in $x3/SE4,
    $x5 in /Source/R3
where $x0/Attr2/text()=$x3/Attr9/text() and
      $x5/Attr14/text()=$x4/Attr12/text() and
      $x5/Attr15/text()=$x4/Attr11/text()
return
    <R7>
      <Attr14> $x5/Attr14/text()
      <Attr15> $x5/Attr15/text()
      for $x20 in /Source/R1,
          $x21 in $x0/SE1, $x22 in $x1/SE2,
          $x23 in /Source/R2,
          $x24 in $x3/SE3, $x24 in $x3/SE4,
          $x25 in /Source/R3
      where
        $x20/Attr2/text()=$x23/Attr9/text() and
        $x25/Attr14/text()=$x24/Attr12/text() and
        $x25/Attr15/text()=$x24/Attr11/text() and
        $x25/Attr14/text()=$x5/Attr14/text() and
        $x25/Attr15/text()=$x5/Attr15/text()
      return
        <SE6>
          <Attr16> $x25/Attr16/text()
          <Attr18> $x20/Attr2/text()
                       * $x25/Attr14/text()
        </SE6>
    </R7>
</Target>
```