Clio: A Semi-Automatic Tool For Schema Mapping

Mauricio A. Hernandez[1] Renee J. Miller[2] Laura M. Haas[1] Lingling Yan[1]
C. T. Howard Ho[1] Xuqing Tian[1]

[1]IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

[2]Department of Computer Science
University of Toronto
6 King's College Rd.
Toronto, ON, Canada M5S 3H5

Abstract

We consider the integration requirements of modern data intensive applications including data warehousing, global information systems and electronic commerce. At the heart of these requirements lies the *schema mapping* problem in which a source (legacy) database must be mapped into a different, but fixed, target schema. The goal of schema mapping is the discovery of a query or set of queries to map source databases into the new structure. We demonstrate *Clio*, a new semi-automated tool for creating schema mappings. Clio employs a mapping-by-example paradigm that relies on the use of *value mappings* describing how a value of a target attribute can be created from a set of values of source attributes.

1 Introduction

Many modern data applications in data warehousing and electronic commerce require merging and coalescing data from multiple diverse sources into a new structure or schema. These applications frequently start with an understanding of how data will be used and viewed in its new schema. For instance, before a data warehouse can be loaded, DBAs and consultants spend months determining what types of queries will be asked, and then designing a schema that will readily support those queries. To load the warehouse, they then must create mappings from the warehouse schema to the underlying data sources' schemas. Creating those mappings is today a largely manual (and extremely complex) process. Likewise, to deploy a global information system, experts first determine what information it will present to the world (with what logical structure), and then create the view definitions that map between the new schema and the data sources. Even simple e-commerce applications often start with a DTD which defines the way data will be presented to users and other applications, and must then map that DTD to the data stored in an underlying source.

Creating such mappings or views entails understanding the relationships between, at least, two schemas: one (or more) source schema(s), and a target schema. *Schema mapping* is in some ways reminiscent of the long-standing problem of schema integration, but with some important distinctions. The classical schema integration methodologies focus on integrating a set of schemas into a unified representation [8]. They treat schema integration as fundamentally a database design problem. Schema mapping, on the other hand, takes the end representation as a given; no design is involved. The focus in schema mapping is, however, on the discovery of the query that realizes the mapping. As with schema integration, the schema mapping task cannot be fully automated since the syntactic representation of schemas and data do not completely convey the semantics of different databases. As a result, for both schema mapping and schema integration, we must rely on an outside source to provide some information about how different schemas (and data) correspond.

Clio is a research prototype of a tool to ease the task of schema mapping [7]. Users load source and target schemas into Clio and graphically connect source objects with target objects. These connections, which we call *value correspondences*, express how the values of one or more source objects are combined and transformed into a target value. For example, a target attribute `salary` may be defined as a function of the values under the source attributes `hours` and `payRate`. Given a set of value correspondences, Clio's main task is deducing a likely schema mapping. This mapping is expressed as SQL view definitions that allow applications to get directly at source data using a middleware query engine. These view definitions can be optimized normally by the query engine, and can be merged with the actual target queries so that only the data needed for a particular query is converted. Clio produces the SQL queries for the user, providing users

with data samples and other feedback to allow them to understand the mappings produced. Data and schema conversions are considered and specified together, and data values can be used to guide the mapping process. Clio handles schematic heterogeneity, and allows complex mappings to be specified quite simply.

We note, however, that given a set of value correspondences, several different schema mappings are possible. Clio systematically generates, ranks, and manages these schema mappings. If needed, Clio asks the user to choose between or to help rank alternative mappings.

Data transformation tools are readily available in the market today (e.g., [2, 3]). As with Clio, many of these data transformation tools allow users to specify correspondences between a source and a target schema. However, these tools typically restrict the user to a single source table and a single source target, or force the user to describe via code fragments exactly how to transform data. Clio primarily differs from these tools in that it allows users to specify value correspondences from multiple source relations into a single target and generates powerful SQL queries (not code) that handle a broad range of complex transformations. These transformations include joins, nested sets, aggregation, and set operations. Further, Clio is able, through appropriate user interaction, to choose among multiple possible join paths and multiple possible query formulations to produce a query that accurately meets the user's need. To the best of our knowledge, ours is the first work that supports understanding and verifying the correctness of complex data transformation queries.

In this demo, we will show how Clio's mapping generation algorithm works and show some practical applications of the tool. The demo will start by illustrating the mapping algorithm on a simple relational-to-relational schema mapping task. By drawing a small number of value correspondences, we will illustrate how alternative mappings are ranked and selected by Clio and how the user can visualize each alternative. We will then explore more complex relational-to-relational and relational-to-XML schema mappings. Throughout the demo we will showcase the tools embedded in Clio that help users explore their data and more efficiently form mappings.

The rest of this proposal is divided as follows. We provide an overview of Clio in Section 2 followed by a brief description of the core mapping algorithm in Section 3. Then in Section 4 we discuss an example scenario that we will use as the basis of our demo.

2 An Overview of Clio

A typical session with Clio starts with the user loading a source and a target schema into the system. These schemas are read from either an underlying Object-Relational database or from an XML file with an associated XML Schema. Users can then draw value correspondences mapping source attributes into target attributes. Clio's mapping engine incrementally produces the SQL queries that realize the mappings implied by the correspondences. Clio provides schema and data browsers and other feedback to allow users to understand the mapping produced.

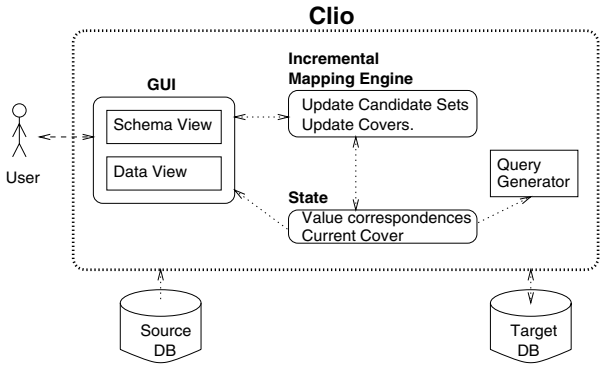

Figure 1: Clio's Architecture

Figure 1 highlights the main components of Clio. A GUI facilitates the process of locating and loading the schemas and entering the value correspondences. Entering and manipulating value correspondences can be done in two modes. In the **Schema View** mode, users see a representation of the source and target schema and create value correspondences by selecting schema objects from the source and mapping them to a target attribute. The alternative **Data View** mode offers a WYSIWYG interface for the mapping process [9] that displays example data for both the source and target tables. Users may add and delete value correspondences from this view and immediately see the changes reflected in the resulting target tuples. Also, the Data View mode helps users navigate through alternative mappings, understanding the often subtle differences between them. For example, in some cases, changing a join from an inner join to an outer join may dramatically change the resulting table. In other cases, the same change may have no effect due to constraints that hold on the source schema. Clio's Data View mode carefully selects target data examples which both illustrate a specific mapping, helping users understand what the mapping does, and which illustrate the differences from any alternative mappings, helping users differentiate mappings.

Both the Schema and Data View directly interact with the component at the heart of Clio: its Incremental Mapping Engine. Clio stores the current mapping as its internal state and, through the incremental algorithm, allows users to move to the next mapping one step at a time. These steps are taken whenever users add, delete or modify a value correspondence. For example, when users add a new value correspondence, the mapping engine will infer and rank alternative mappings that can be formed with the new mapping. The top-ranked alternative will be suggested

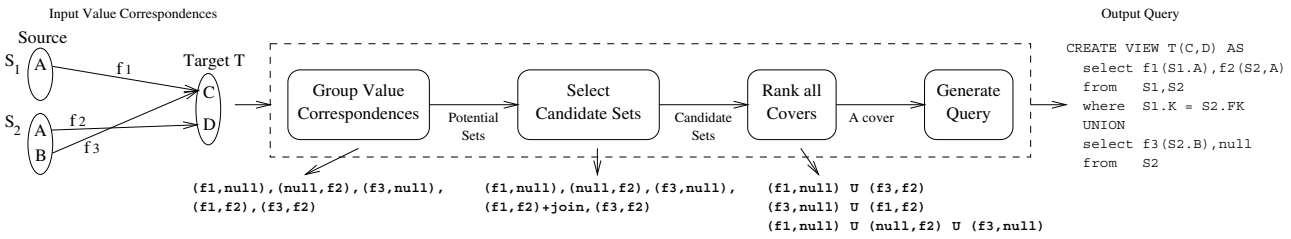

Figure 2: Mapping Construction Algorithm

first and displayed on the GUI. Users can, of course, examine and choose another mapping from the set of alternatives. We highlight some details of the mapping algorithm in the next section.

Although the demo will concentrate on illustrating the use of the Schema View and the Data View to guide mappings, two auxiliary modules will be used as part of the demo. One module mines source data for keys and foreign key constraints and the other uses source and target data to find "matching" attributes.

Whenever the user specifies a value correspondence that maps data from multiple source relations into a target attribute, Clio must search for ways of joining those sources and present the alternatives (if any) as a ranked list of potential join paths. The module that finds connections (join paths) between the source relations is key to the overall show of "intelligence" projected by Clio. Without this module, Clio's front-end is reduced to a "query-by-example" interface in which users explicitly tell the system how to join the source tables. One simple way of gathering potential connections between sources is by reading the key/foreign key (referential integrity) constraints from the database catalog. Since key/foreign key pairs frequently capture the relationships among the entities modeled by the tables, Clio can suggest them as likely connections between source tables. However, it is often the case that information on key and foreign key constraints is either missing from the catalog or incomplete. For this reason, Clio includes tools that mine source data for key and foreign key constraints. These algorithms heuristically search for keys using a sample of the source relations. The discovered keys then seed another search for the foreign keys that refer to those keys.

The second module, the *attribute matching* module, uses some data mining techniques to compare source and target attributes. This module uses a novel variation of existing classification techniques based on domain-independent feature selection [5]. Specifically, a feature vector is computed for each source and target attribute characterizing the overall signature of the data under that attribute. These vectors are then used to match and cluster attributes with similar data signatures. There are two key places where Clio takes advantage of this module. Consider the case in which a mapping is desired between two tables, each with a large number of attributes. As the user enters value correspondences between those two tables, Clio suggests other "obvious" (according to the attribute match) value correspondences between those tables. When the number of source (and target) tables is large, the user is faced with the tedious task of bringing into view relevant tables manually. To alleviate this burden, we have extended the attribute matching module to match entire tables. Upon demand, Clio suggests a number of source tables that are "close" or similar to the currently selected target table and brings them into view automatically.

3 Query Discovery Algorithm

We now present our mapping construction algorithm. To keep the notation simple, we assume the source and target schemas are represented in the relational model. We note that Clio implements an incremental version of this algorithm with support for aggregate functions and nested sets [7].

We divide the algorithm's tasks into four phases (see Figure 2). In the first phase, the value correspondences in $\mathcal{V}$ are partitioned into sets $\{c_1, \ldots, c_p\}$ that contain *at most one* correspondence per attribute of T. We call each such set a *potential candidate set*. In essence, each c_j represents one possible way of mapping the attributes of T. For instance, in Figure 2, one way of computing the target T is by using `f1` for $T.C$ and *null* for $T.D$[1]. Another way is using `f1` for $T.C$ and `f2` for $T.D$. Potential candidate sets are not necessarily disjoint since the same value mapping can appear in multiple potential candidate sets.

In the second phase of the algorithm, we prune from the set of potential candidate sets those sets that cannot be mapped into a good query. In particular, if the value correspondences in the potential candidate set map values from several source relations, we need to find a way of joining the tuples of those relations. If no acceptable join path can be found, the potential candidate set is removed from further consideration. The result of this second phase is a set of *candidate sets*. Value correspondences in a candidate set either map attributes from only one source relation, or map attributes from multiple source relations and a join path among those relations is known.

[1] Represented `(f1, null)` in Figure 2.

In the third phase of the algorithm, we attempt to find a subset Γ of the candidate sets that *covers* all value correspondences in $\mathcal{V}$ (that is, every value correspondence in $\mathcal{V}$ appears at least once in Γ). We permit correspondences to participate in multiple candidates within a cover, but we do not consider a set of candidates Γ if we can remove a candidate set and still have a cover. Following the example in Figure 2, one cover is formed by grouping `(f1,null)` and `(f3, f2)` together since this cover includes all three value correspondences. If there is more than one cover, Clio ranks them in reverse order of the number of candidate sets in the cover. Since the number of candidates in a cover is the number of candidate SQL queries needed to compute the mapping, we prefer smaller covers which will produce simpler mappings. When two or more covers have the same number of candidate sets, we prefer those that use the largest number of target attributes in all candidate sets and, thus, minimize the number of "null" values in the target. The ranked covers are presented as alternative mappings for the user to evaluate.

The final step is to build the query from the selected cover. For each candidate set c_j in the selected cover, we create a candidate SQL query such that all correspondence function in c_j appear in the `SELECT` clause, all source relations are mentioned in the `FROM` clause, and all user-defined predicates appear as a conjunction in the `WHERE` clause. Any join path determined in the second step for this candidate set will be used to determine the appropriate source relations for the `FROM` clause. The join predicates are also added to the `WHERE` clause. All candidate SQL queries are then combined into one large query using the multiset `UNION`. In Figure 2, the resulting query corresponds to the second cover, `(f3,null) UNION (f1, f2)`.

4 A Data Warehouse Example

We have at our disposition a number of data sources that can be used to illustrate Clio. In this proposal, we present an example based on a proposed software engineering warehouse for storing and exchanging information extracted from computer programs [1]. Such warehouses have been proposed both to enable new program analysis applications, including data mining applications [6], and to promote data exchange between research groups using different tools and software artifacts for experimentation [4]. Figure 3 depicts a portion of a warehouse schema for this information. This schema has been designed to represent data about a diverse collection of software artifacts that have been extracted using different software analysis tools. The warehouse schema was designed to be as flexible as possible. As a result, it uses a very generic representation of software data as labeled multi-graphs. Conceptually, software artifacts (for example, functions, data types, macros, *etc.*) form the nodes of the graph. Associations or references between artifacts (for example, function calls or data references) form the edges. Two of the main tables for artifacts and references are depicted in the figure.

As new software analysis tools are developed, the data from these tools must be mapped into this integrated schema. In Figure 3, we also give a relational representation of an example source schema. Foreign keys are depicted by dashed lines. (We will demonstrate how Clio uses data to mine for these key/foreign key pairs.) To map the source data into the warehouse, the correspondences of Figure 4 may be used. Note that correspondences f_4 and f_5 map the relation name into the ReferenceType value, effectively transforming schema to data.

The grouping algorithm of Clio uses the foreign key information in the source to create several candidate sets. One contains the four correspondences $\{f_1, f_2, f_3, f_4\}$. Note that there are two foreign key join paths between the source relations involved in these correspondences. The first populates the `Source` attribute of the target with the `File` attribute of the caller function (Mapping S_1). The second populates the `Source` attribute of the target with the `File` attribute of the called function (Mapping S_2).

S_1:
```
SELECT C.Caller, C.Callee, relname(C), F.File
FROM   Calls C, Function F
WHERE  C.Caller = F.Name
```

S_2:
```
SELECT C.Caller, C.Callee, relname(C), F.File
FROM   Calls C, Function F
WHERE  C.Callee = F.Name
```

If we cannot distinguish these paths using the data, both will be presented to the user. The user may use the Data View mode and examine example values for both candidate mappings to evaluate which of the join paths is correct. Based on the data, the user can pick the desired mapping. In this example, the user would choose the first since the source location of a program call is the location of the caller function.

A second candidate subset contains the four correspondences $\{f_5, f_6, f_7, f_3\}$. Note that Clio chooses to use f_3 in both candidates since there is a good foreign key path to use for both candidates. These two candidate sets form a cover. Clio combines the Mapping S_1 and the mapping produced for this second candidate subset to produce the following complete schema mapping. Since Clio favors grouping correspondences from the same relation, the other covers possible in this example are eliminated.

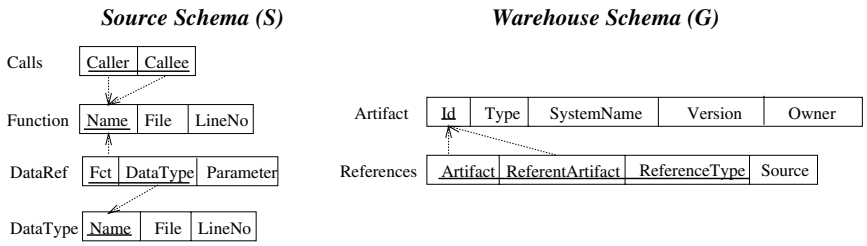

Figure 3: Schema of the source database and the Software Engineering Warehouse

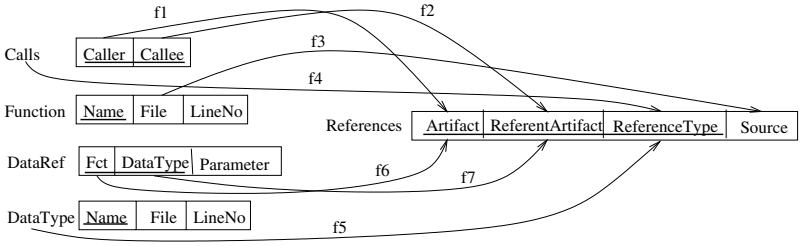

Figure 4: Value correspondences used to map the source schema to the Warehouse

```
S: SELECT C.Caller, C.Callee, relname(C), F.File
   FROM   Calls C, Function F
   WHERE  C.Caller = F.Name
   UNION ALL
   SELECT D.Fct, D.DataType, relname(D), F.File
   FROM   DataRef D, Function F
   WHERE  D.Fct = F.Name
```

To extend this example, consider the correspondence and filter used to define the `Class` table (a subtable of the `Artifact` table for storing information about object-oriented classes). For brevity, we assume the `Class` table has a single `Id` attribute indicating the data type of the class (Figure 5).

The correspondence f_a maps data types to the `Class` table. The user also provides a filter p_a restricting the mapping to data types referenced by more than 5 functions. Clio discovers the join paths between `DataRef` and `DataType`. Given the aggregate function in the filter, the discovered mapping includes a group by and is depicted below.

```
S_a: SELECT T.Name
     FROM   DataType T, DataRef R
     WHERE  T.Name = R.DataType
     GROUP BY T.Name
     HAVING count(R.Fct) > 5
```

In addition to these examples, the demo will include an example in which we map this example source schema into an XML document, producing the necessary nested queries. We will demonstrate how a set of simple value correspondences lead to a complex nested query further illustrating the need for tools like Clio.

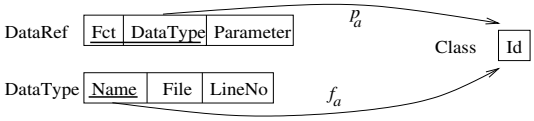

Figure 5: Aggregate filter in a value correspondence.

References

[1] I. Bowman, M. Godfrey, and R. Holt. Connecting Software Architecture Recovery Frameworks. In *Proc. of the First Int'l Symp. on Constructing Soft. Eng. Tools (CoSET'99)*, Los Angeles, May 17-18 1999.

[2] Data Junction. http://www.datajunction.com.

[3] ETI. ETI*EXTRACT. http://www.eti.com.

[4] M. Harrold, R. J. Miller, A. Porter, and G. Rothermel. A Collaborative Investigation of Program-Analysis-Based Testing and Maintenance. In *International Workshop on Experimental Studies of Software Maintenance*, pages 51–56, Bari, Italy, October 1997.

[5] C. T. H. Ho and X. Tian. Automatic Classification of Database Columns Using Feature Analysis. Submitted for consideration to ACM SIGMOD 2001.

[6] R. J. Miller and A. Gujarathi. Mining for Program Structure. *Int'l Journal on Software Eng. and Knowledge Eng.*, 9(5):499–517, 1999.

[7] R. J. Miller, L. M. Haas, and M. Hernández. Schema Mapping as Query Discovery. In *Proc. of the Int'l Conf. on VLDB*, Cairo, Egypt, 2000.

[8] S. Ram and V. Ramesh. Schema Integration: Past, Current and Future. In A. Elmagarmid, M. Rusinkiewicz, and A. Sheth, editors, *Management of Heterogeneous and Autonomous Database Systems*, pages 119–155. Morgan Kaufmann Publishers, 1999.

[9] L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. Submitted for consideration to ACM SIGMOD 2001.