Clio Grows Up: From Research Prototype to Industrial Tool

Laura M. Haas
IBM Silicon Valley Labs
laura@almaden.ibm.com

Mauricio A. Hernández
IBM Almaden Research Center
mauricio@almaden.ibm.com

Howard Ho
IBM Almaden Research Center
ho@almaden.ibm.com

Lucian Popa
IBM Almaden Research Center
lucian@almaden.ibm.com

Mary Roth
IBM Silicon Valley Labs
torkroth@us.ibm.com

ABSTRACT

Clio, the IBM Research system for expressing declarative schema mappings, has progressed in the past few years from a research prototype into a technology that is behind some of IBM's mapping technology. Clio provides a declarative way of specifying schema mappings between either XML or relational schemas. Mappings are compiled into an abstract query graph representation that captures the transformation semantics of the mappings. The query graph can then be serialized into different query languages, depending on the kind of schemas and systems involved in the mapping. Clio currently produces XQuery, XSLT, SQL, and SQL/XML queries. In this paper, we revisit the architecture and algorithms behind Clio. We then discuss some implementation issues, optimizations needed for scalability, and general lessons learned in the road towards creating an industrial-strength tool.

1. INTRODUCTION

Mappings between different representations of data are fundamental for applications that require data interoperability, that is, integration and exchange of data residing at multiple sites, in different formats (or schemas), and even under different data models (such as relational or XML). To provide interoperability, information integration systems must be able to understand and translate between the various ways in which data is structured. With the advent of the flexible XML format, the abundance of different schemas describing similar or related data has proliferated even more.

We can distinguish between two main forms of data interoperability. *Data exchange* (or *data translation*) is the task of restructuring data from a source format (or schema) into a target format (or schema). This is not a new problem; the first systems supporting the restructuring and translation of data were built several decades ago. An early such system was EXPRESS [9], which performed data exchange between hierarchical schemas. However, the need for systems supporting data exchange has persisted and, in fact, grew larger over the years. Data exchange requirements appear in the ETL (extract-transform-load) workflows, used to populate a data warehouse from a set of data sources, in XML messaging, in schema evolution (when migrating data from an old version to a new version), in database restructuring, etc. A second form of data interoperability is *data integration* (or *federation*), which means the ability to query a set of heterogeneous data sources via a virtual unified target schema. There is no need to materialize a target instance in this case; instead, the emphasis is on query processing.

In both cases, of data exchange and data integration, relationships or mappings must first be established between the source schema(s) and the target schema. There are two complementary levels at which mappings between schemas can be established, and both pose challenges. The first is mostly a syntactic one: by employing *schema matching* techniques, a set of uninterpreted *correspondences* between elements (terms, field names, etc.) in two different schemas are established. A significant body of work on schema matching algorithms has been developed (see [8] for a survey). The second level at which mappings can be established is a more operational one, that can relate *instances* over the source schema(s) with *instances* over the target schema. Establishing an "operational" mapping is necessary if one needs to move actual data from a source to a target, or to answer queries. Such "operational" mappings can be seen as interpretations (with runtime consequences) of the correspondences that result after schema matching. It is this second view on schema mappings that we focus on here, and we will use the term schema mappings to mean "operational" mappings rather than correspondences.

Schema mappings in various logical forms have been used for query answering in data integration (see [4] for a survey). Schema mappings expressed as constraints (source-to-target tgds) have been used to formalize data exchange between relational schemas [3]. Similar schema mappings appear as an important component of the model management framework of Bernstein et al [1, 5].

The system that we are about to describe, *Clio*, is the first to address the problem of semi-automatic generation of schema mappings as well as the subsequent problem of using the schema mappings for the actual runtime (e.g., how to generate an XQuery or XSLT transformation from the schema mappings to implement the data exchange). Originally designed for the relational case [6], Clio has then evolved into a full-fledged system [7] for generating mappings and transformations between hierarchical XML schemas. In this paper, we describe our further experience in building an industrial-strength Clio; in particular, we focus on the practical challenges encountered in: (1) the scalable semi-automatic generation of mappings between schemas with large degree of complexity, and (2) the subsequent use of these schema mappings to accomplish efficient and functional data exchange, via query generation.

Clio system architecture. A pictorial view that shows how Clio is structured into its main components is shown in Figure 1. At the core of the system are the *mapping generation* component and the *query generation* component. The mapping generation component takes as input correspondences between the source and the target schemas and generates a schema mapping consisting of a *set of logical mappings* that provide an interpretation of the given correspondences. The logical mappings are declarative assertions (expressed as source-to-target constraints, to be described shortly). They can be viewed as abstractions (or requirements) for the more complex *physical* transformations (e.g., SQL, XQuery or XSLT scripts) that operate at the data

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.

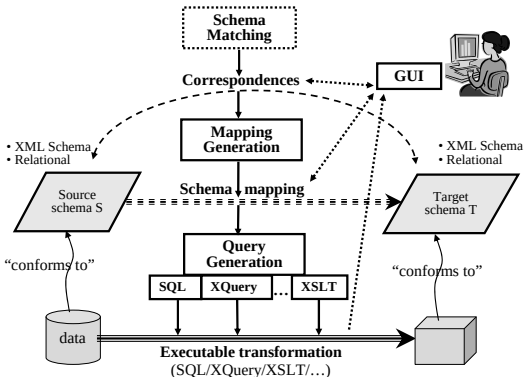

Figure 1: Clio architecture

transformation runtime. Such abstractions are easier to understand and to reason about, and are independent of any physical transformation language. Nonetheless, they capture most of the information that is needed to generate the physical artifacts automatically. The query generation component has then the role to convert a set of logical mappings into an executable transformation script. Query generation consists of a generic module, independent of a particular execution language, and of a number of pluggable components that are specific to each execution language: SQL, SQL/XML, XQuery and XSLT. At any time during the design, a user can interact with the system through a *GUI component*. The user can view, add and remove correspondences between the schemas, can attach transformation functions to such correspondences, can inspect and edit (in a controlled way) the generated logical mappings, and can inspect (without editing) the generated transformation script. Finally, correspondences can also be generated via an optional *schema matching* component. Clio can interact with any schema matcher and also has its own built-in schema matching algorithm. However, in this paper, we will focus on the rest of the components, which are specific to Clio.

2. MAPPING AND QUERY GENERATION

Figure 2 illustrates an actual Clio screenshot showing portions of two gene expression schemas and correspondences between these schemas. We will use this as a running example to illustrate the concepts, the algorithms, as well as the challenges that we faced in designing the mapping and query generation components.

The left-hand (source) schema, GENEX, is a relational schema for gene expression (microarray) data stored in the GeneX database[1]. It is a schema of medium complexity; it consists of 63 tables that are inter-related via key and foreign key constraints (there are 47 foreign key constraints that encode the relationships between tables). The right-hand (target) schema is an XML schema (GeneXML, formerly GEML) intended for the exchange (with other databases) of gene expression data. In addition to a structural change to a hierarchical format, this schema also presents changes in the concrete elements and attributes (some of the source GENEX elements do not have correspondences in GeneXML and vice-versa). The XML schema is 24KB in size, including 286 elements and attributes. The number of foreign keys (keyref) is much reduced compared to the relational GENEX schema, since many of the original foreign keys are now replaced by hierarchical parent-child relationships.

For illustration purposes, in this section we only describe the process of mapping data about experiment sets and about the factors and treatments that were applied to each experiment set. Thus, we ignore a large portion of the two schemas and focus only on the relevant tables and XML structures (shown in Figure 2). To generate a complete transformation, the remainders of the schemas (e.g. samples, array measurements, citations, software, etc.) will also have to be

[1]http://sourceforge.net/projects/genex

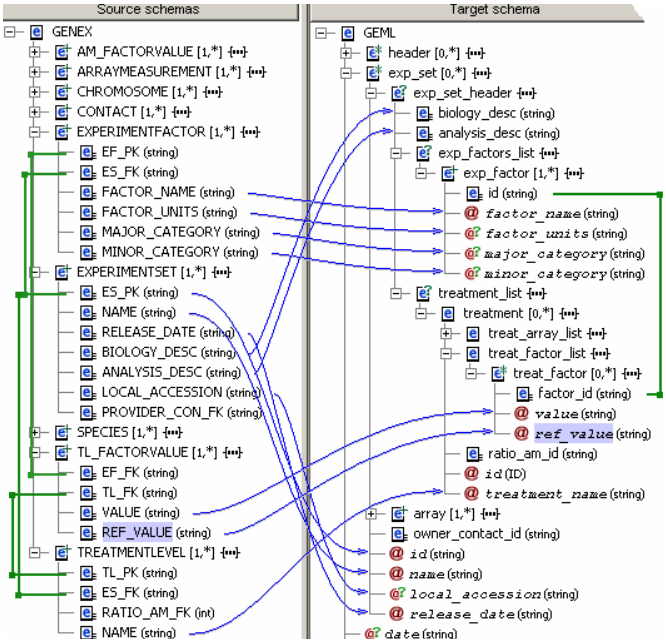

Figure 2: Schemas and correspondences in Clio.

matched; additional mappings will then be generated and the final result will be materialized into a complete transformation script. In Section 3 we give further details on the complexity and challenges of generating a large-scale transformation.

Schemas in Clio The first step in Clio is loading and transforming the schemas into an internal representation. The model that we adopt for this purpose is a nested relational model, that is suitable for describing relational tables as well as the XML hierarchy. In its basic form, the nested relational model consists of several fundamental types: set, record and choice types, together with atomic (primitive) types. For example, `GEML` in Figure 2 is represented as a record type and its repeatable element `exp_set` is represented via a set type. In addition to this basic nested relational model, Clio's implementation of a schema loader includes a series of add-on features to capture the intricacies of XML Schema: data fields to remember whether a record component corresponds to an element or an attribute and whether the component is optional or nillable, various encodings to deal with mixed content elements, derived types, type variables (used, in particular, to encode recursive types), etc.

2.1 Mapping Generation

The main idea of the algorithm [7] is to put correspondences in groups (not necessarily disjoint) based on how their end-points (source, respectively, target) relate to each other in the schemas. Thus, for any two *related* elements in the source schema, for which there exist correspondences into two *related* elements in the target schema, there will be a group (and, subsequently, a mapping) that includes those correspondences (and possibly more). As an example, `FACTOR_NAME` and `BIOLOGY_DESC` are related in the source because there is a foreign key that links the `EXPERIMENTFACTOR` table to the `EXPERIMENTSET` table); furthermore, `biology_desc` and `factor_name` are related in the target because the latter is a child of the former. Hence, there will be a mapping that maps related instances of `BIOLOGY_DESC` and `FACTOR_NAME` (and possibly more) into related instances of `biology_desc` and `factor_name` (and possibly more).

Generation of tableaux The first step of the algorithm is to generate all the basic ways in which elements relate to each other within one schema, based on the schema structure and constraints. This generation step (described in [7] in detail) considers each set-type element in a nested schema as a direct generalization of the concept of a table in a relational schema; it then joins to each such set-type element all

S_1 = ExpSet
S_2 = ExpFactor $\bowtie_{ES\_FK = ES\_PK}$ ExpSet
S_3 = TreatmentLevel $\bowtie_{ES\_FK = ES\_PK}$ ExpSet
S_4 = TL_FactorValue $\bowtie_{TL\_FK = TL\_PK}$ TreatmentLevel $\bowtie_{ES\_FK = ES\_PK}$ ExpSet
$\searrow \bowtie_{EF\_FK = EF\_PK}$ ExpFactor $\bowtie_{ES\_FK = ES\_PK}$ ExpSet

T_1 = GEML/exp_set
T_2 = GEML/exp_set/…/exp_factor
T_3 = GEML/exp_set/…/treatment
T_4 = GEML/exp_set/ { …/treatment/…/treat_factor $\bowtie_{factor\_id = id}$ …/exp_factor }

Figure 3: Source and target tableaux (informal notation).

the other set-type elements that can be reached by following foreign key (keyref) constraints (a process called the *chase*). The result is a set of *tableaux*[2], one set in each schema.

In Figure 3 we show several of the source and target tableaux that are generated for our example. (For brevity, we sometimes use `ExpSet` instead of `EXPERIMENTSET`; similar abbreviations are also used for the other tables.) For the source schema, `ExpSet` forms a tableau by itself, because `ExpSet` is a top-level table (there are no outgoing foreign keys). In contrast, `ExpFactor` does not form a tableau by itself but needs `ExpSet` into which it has a foreign key. A more complicated tableau is S_4 that is constructed for `TL_FactorValue`. Each factor value is associated to one treatment level (thus, the foreign key into `TreatmentLevel`) and each treatment level corresponds to one experiment set (thus, the foreign key into `ExpSet`). However, a factor value is also an experiment factor (thus, the foreign key into `ExpFactor`), and each experiment factor is associated to an experiment set (hence, the second occurrence of `ExpSet`).

The above tableau S_4 illustrates the complexity that can arise even when a relatively small number of tables is involved. An additional constraint (which is true for the actual data) could be used to infer that the two occurrences of `ExpSet` correspond to the same experiment set instance. Clio's chasing engine includes such an inference mechanism. However, such a constraint is hard to extract in practice (it is not a key, but a complicated dependency). The price to pay for not having such a constraint will be further ambiguity that remains to be solved during mapping generation (to be described shortly).

For the target schema, the tableaux are more complex, due to nesting and also to the fact that predicates (such as join conditions) can have a context. While tableaux T_1, T_2 and T_3 are straight paths to set-type elements (e.g., `exp_factor`), the tableau T_4, intended to denote the collection of treatment factors `treat_factor`, also contains a join with `exp_factor`. Moreover, the join is relative to a given instance of `exp_set`. The reason for this is the existence of a keyref constraint that associates every `treat_factor` element with an `exp_factor` element, within the *same* `exp_set` instance. Such keyref constraints, part of the XML Schema specification, can be easily specified by putting the constraint on the correct element in the hierarchy (`exp_set` instead of the root, for this example).

To represent tableaux such as T_4 and S_4 unambiguously, Clio uses an internal notation based on: (1) generators, which are used to bind variables to individual elements in sets, and (2) conditions. Path expressions, which can be absolute or relative to the bound variables, can appear in both the generators and the conditions. As an example, the internal form of T_4 is shown below.

Generators:
$t_0 \in$ GEML/exp_set, $t_1 \in t_0$/exp_set_header/treatment_list/treatment,
$t_2 \in t_1$/treat_factor_list/treat_factor,
$t_3 \in t_0$/exp_set_header/exp_factor_list/exp_factor ;
Conditions:
t_2/factor_id = t_3/id

The use of collection-bound variables (rather than that of arbitrarily bound variables) has the advantage that the resulting notation

[2]also called *logical relations* in [7]. However, the term *tableaux*, used in classical papers on the chase, was the one widely used during the development of Clio. Hence, we stand by it here.

1. $\mathbf{m_1}$: $\forall$ $\mathbf{s_0} \in$ /ExpFactor, $\mathbf{s_1} \in$ /ExpSet where $\mathbf{s_0}$/es_fk = $\mathbf{s_1}$/es_pk
2. $\exists$ $\mathbf{t_0} \in$ /GEML/exp_set, $\mathbf{t_1} \in \mathbf{t_0}$/exp_set_header/exp_factor_list/exp_factor
3. such that $\mathbf{s_1}$/LOCAL_ACCESSION = $\mathbf{t_0}$ /@local_accession $\wedge$
4. $\mathbf{s_1}$/NAME = $\mathbf{t_0}$/@name $\wedge$ $\mathbf{s_1}$/ES_PK = $\mathbf{t_0}$/@id $\wedge$
5. $\mathbf{s_1}$/RELEASE_DATE = $\mathbf{t_0}$/@release_date $\wedge$
6. $\mathbf{s_1}$/ANALYSIS_DESC = $\mathbf{t_0}$/exp_set_header/analysis_desc $\wedge$
…

1. $\mathbf{m_3}$: $\forall$ $\mathbf{s_0} \in$ /TL_FactorValue, $\mathbf{s_1} \in$ /TreatmentLevel, $\mathbf{s_2} \in$ /ExpSet,
2. $\mathbf{s_3} \in$ /ExpFactor, $\mathbf{s_4} \in$ /ExpSet,
3. where $\mathbf{s_0}$/tl_fk = $\mathbf{s_1}$/tl_pk $\wedge$ $\mathbf{s_1}$/es_fk = $\mathbf{s_2}$/es_pk $\wedge$
4. $\mathbf{s_0}$/ef_fk = $\mathbf{s_3}$/ef_pk $\wedge$ $\mathbf{s_3}$/es_fk = $\mathbf{s_4}$/es_pk
5. $\exists$ $\mathbf{t_0} \in$ /GEML/exp_set, $\mathbf{t_1} \in \mathbf{t_0}$/exp_set_header/treatment_list/treatment,
6. $\mathbf{t_2} \in \mathbf{t_1}$/treat_factor_list/treat_factor, $\mathbf{t_3} \in \mathbf{t_0}$/exp_set_header/exp_factor_list/exp_factor
7. where $\mathbf{t_2}$/factor_id = $\mathbf{t_3}$/id
8. such that …

Figure 4: Two logical mappings.

maps easily to *efficient* iteration patterns (e.g., the from clause of SQL, or the for loops of XQuery/XSLT).

Generation of logical mappings The second step of the mapping generation algorithm is the generation of logical mappings. (Recall that a schema mapping is a set of logical mappings.) The basic algorithm in [7] pairs all the existing tableaux in the source with all the existing tableaux in the target, and then finds the correspondences that are covered by each pair. If there are such correspondences, then the given pair of tableaux is a candidate of a logical mapping. (In Section 3 we describe an additional filtering that takes place before such a candidate is actually output to a user.)

In Figure 4 we show two of the resulting logical mappings, for our example. The mapping m_1 is obtained from the pair (S_2, T_2), which is covered by 10 correspondences. The source tableau is encoded in the $\forall$ clause (with its associated where clause that stores the conditions). A similar encoding happens for the target tableau, except that an $\exists$ clause is used instead of $\forall$. Finally, the such that clause encodes all the correspondences between the source and the target that are covered. (Only five of them are shown in the figure.)

A more complex mapping is m_3 that is obtained from the pair (S_4, T_4), which is covered by all the correspondences shown in Figure 2. An additional complication in generating this mapping arises from the fact that the correspondences that map `EXPERIMENTSET` columns to `exp_set` elements/attributes have multiple interpretations, because each correspondence can match either the first occurrence or the second occurrence of `ExpSet` in S_4. This ambiguity is resolved by generating all possible interpretations (e.g., all these correspondences match the first occurrence of `ExpSet`, or all match the second occurrence, or some match the first occurrence and some match the second occurrence). A user would then have to go through all these choices and select the desired semantics. The default choice that we provide is the one in which all the ambiguous correspondences match the first choice (e.g., the first occurrence of `ExpSet`). For this example, all the different interpretations are in fact equivalent, since the two occurrences of `ExpSet` represent the same instance, due to the constraint discussed earlier.

In Clio, the tableaux that are constructed based on chasing form only the basic (default) way of constructing mappings. Users have the option of creating additional mappings through the mapping editor. In each schema, a new tableau can be specified by selecting the needed collections and then creating predicates on those collections. Each such tableau can then participate in the algorithm for mapping generation. In other words, each tableau will be paired with all the tableaux in the opposite schema to reach all possible mappings, based on covered correspondences.

Mapping language To the language that we have just described, we must add an additional construct: *Skolem functions*. These functions can explicitly represent target elements for which no source value is given. For example, the mapping m_1 of Figure 4 will not specify a value for the `@id` attribute under `exp_factor` (because there is no correspondence to map into `@id`). To create a unique value for this attribute, which is required by the target schema, a Skolem func-

```
1. for $x0 in $doc0/GENEX/EXPERIMENTFACTOR,
2.     $x1 in $doc0/GENEX/EXPERIMENTSET
3. where
4.   $x1/ES_PK/text() = $x0/ES_FK/text()
5. return
6.   <exp_set>
7.    { attribute id { $x1/ES_PK } }
8.    { attribute name { $x1/NAME } }
9.    { attribute local_accession { $x1/LOCAL_ACCESSION } }
10.   { attribute release_date { $x1/RELEASE_DATE } }
11.   <exp_set_header>
12.     <biology_desc> { $x1/BIOLOGY_DESC/text() } </biology_desc>
13.     <analysis_desc> { $x1/ANALYSIS_DESC/text() } </analysis_desc>
14.     <exp_factors_list> {
15        for $x0L1 IN $doc0/GENEX/EXPERIMENTFACTOR,
16.           $x1L1 IN $doc0/GENEX/EXPERIMENTSET
17.       where
18.         $x1L1/ES_PK/text() = $x0L1/ES_FK/text() and
19.         $x1/BIOLOGY_DESC/text() = $x1L1/BIOLOGY_DESC/text() and
20.         $x1/ANALYSIS_DESC/text() = $x1L1/ANALYSIS_DESC/text() and
21.         $x1/NAME/text() = $x1L1/NAME/text() and
22.         $x1/LOCAL_ACCESSION/text() = $x1L1/LOCAL_ACCESSION/text() and
23.         $x1/RELEASE_DATE/text() = $x1L1/RELEASE_DATE/text()
24.        return
25.         <exp_factor>
26.           { attribute factor_name { $x0L1/FACTOR_NAME } }
27.           { attribute factor_units { $x0L1/FACTOR_UNITS } }
28.           { attribute major_category { $x0L1/MAJOR_CATEGORY } }
29.           { attribute minor_category { $x0L1/MINOR_CATEGORY } }
30.           <id>
        {"SK6(",$x1L1/BIOLOGY_DESC/text(),$x1L1/ANALYSIS_DESC/text(),…}</id>
31.         </exp_factor>
32.     } </exp_factors_list>
33.   </exp_set_header>
34. </exp_set>
```

Figure 5: XQuery fragment for one of the logical mappings.

tion can be generated (and the siblings and ancestors of `@id`, such as `factor_name` and `biology_desc`, which contain a source value, will appear as arguments of the function). In general, the generation of Skolem functions could be postponed until query generation (see Section 2.2) and the schema mapping language itself could avoid Skolem functions. However, to be able to express mappings that arise from *mapping composition* (e.g., a mapping that is equivalent to the sequence of two consecutive schema mappings), functions *must* be part of the language [2]. Clio has been recently extended to support mapping composition, an important feature of metadata management. Hence, the schema mapping language used in Clio includes Skolem functions; the resulting language is a nested relational variation of the language of second-order tgds of Fagin et al [2].

2.2 Query Generation

Our basic query generation operates as follows. Each logical mapping is compiled into a query graph that encodes how each target element/attribute is populated from the source-side data. For each logical mapping, query generators walk the relevant part of the target schema and create the necessary join and grouping conditions. The query graph also includes information on what source data-values each target element/attribute depends on. This is used to generate unique values for required target elements as well as for grouping. In Figure 5, we show the XQuery fragment that produces the target `<exp_set>` elements as prescribed by logical mapping m_1[3]. The query graph encodes that an `<exp_set>` element will be generated for every tuple produced by the join of the source tables `EXPERIMENTFACTOR` and `EXPERIMENTSET` (see lines 1–2 in m_1). Lines 1–4 in Figure 5 implement this join. Lines 7–10 output the attributes within `exp_set` and implement lines 3–5 of m_1. Then, we need to produce the repeatable `exp_factor` elements. The query graph prescribes two things about `exp_factor`: 1) that it will be generated for every tuple that results from the join of `EXPERIMENTFACTOR` and `EXPERIMENTSET` (lines 15–18 in the query – they are the same as 1–4 except for the variable renaming), and, 2) since it appears nested within `exp_set`, that such tuples join with the current tuple from the outer part of the query to create the proper grouping (lines 19–23 – requiring that the values for `exp_set` in the inner tuple be the same as in the outer tuple). The grouping condition in lines 19–23 does not appear anywhere in logical mapping m_1. This is computed in the query graph when the shape of the target schema is taken into consideration. Finally, lines 25–31 produce an actual `exp_factor` element. Of particular interest, line 30 creates a value for the `id` element. No correspondence exists for the `id` element and, thus, there is no value for it in m_1. However, since `id` is a *required* target element, the query generator produces a Skolem value for `id` based on the dependency information stored in the query graph.

3. PRACTICAL CHALLENGES

In this section, we present several of the implementation and optimization techniques that we developed in order to address some of the many challenging issues in mapping and query generation.

3.1 Scalable Incremental Mapping Generation

One of the features of the basic mapping generation algorithm is that it enumerates a priori all possible "skeletons" of mappings, that is, pairs of source and target tableaux. In a second phase (insertion/deletion of correspondences), mappings are generated, based on the precomputed tableaux, as correspondences are added in or removed. This second phase must be *incremental* in that, after the insertion of a correspondence (or of a batch of correspondences) or after the removal of a correspondence, a new mapping state must be efficiently computed based on the previous mapping state. This is an important requirement since one of the main uses of Clio is interactive mapping generation and editing. To achieve this, the main data structure in Clio (the mapping state) is the *list of skeletons*, where a skeleton is a pair of tableaux (source and target) together with all the inserted correspondences that match the given pair of tableaux. When correspondences are inserted or deleted, the relevant skeletons are updated. Furthermore, at any given state, only a subset of all the skeletons is used to generate mappings. The other skeletons are deemed *redundant* (although they could become non-redundant as more correspondences are added[4]). This redundancy check, which we describe next, can significantly reduce the amount of irrelevant mappings that a user has to go through.

Redundancy check A tableau T_1 is a *sub-tableau* of a tableau T_2 if there is a one-to-one mapping of the variables of T_1 into the variables of T_2, so that all the generators and all the conditions of T_1 become respective subsets of the generators and conditions of T_2. A pair (S_i, T_j) of source and target tableaux is a *sub-skeleton* of a similar pair (S'_i, T'_j) if S_i is a sub-tableau of S'_i and T_j is a sub-tableau of T'_j. The sub-tableaux relationships in each of the two schemas as well as the resulting sub-skeleton relationship are also precomputed in the first phase, to speed up the subsequent processing that occurs in the second phase. When correspondences are added, in the second phase, if (S_i, T_j) is a sub-skeleton of (S'_i, T'_j) and the set C of correspondences covered by (S'_i, T'_j) is the *same* as the set of correspondences covered by (S_i, T_j), then the mapping based on (S'_i, T'_j) and C is redundant. Intuitively, we do not want to use large tableaux unless more correspondences will be covered. For example, suppose that we have inserted only two correspondences, from `BIOLOGY_DESC` and `ANALYSIS_DESC` of `EXPERIMENTSET` to `biology_desc` and `analysis_desc` under `exp_set`. These two correspondences match on the skeleton (S_2, T_2) where S_2 and T_2 are the tableaux in Figure 3 involving experiment sets and experiment factors. However, this skeleton and the resulting mapping are redundant, because there is a sub-skeleton (S_1, T_1) that covers

[3]The complete XQuery is the concatenation of all the fragments that correspond to all the logical mappings.

[4]And vice-versa, non-redundant skeletons can become redundant as correspondences are removed.

the same correspondences, where S_1 and T_1 are the tableaux in Figure 3 involving experiment sets. Until a correspondence that maps EXPERIMENTFACTOR is added, there is no need to generate a logical mapping that involves EXPERIMENTFACTOR.

The hybrid algorithm The separation of mapping generation into the two phases (precomputation of tableaux and then insertion of correspondences) has one major advantage: when correspondences are added, no time is spent on finding associations between the elements being mapped. All the basic associations are already computed and encoded in the tableaux; the correspondences are just matched on the relevant skeletons, thus speeding up the user addition of correspondences in the GUI. There is also a downside: when schemas are large, the number of tableaux can also become large. The number of skeletons (which is the product of the numbers of source and, respectively, target tableaux) is even larger. A significant amount of time is then spent in the preprocessing phase to compute the tableaux and skeletons as well as the sub-tableaux and sub-skeleton relationships. A large amount of memory may be needed to hold all the data structures. Moreover, some of the skeletons may not be needed (or at least not until some correspondence is added that matches them).

A more scalable solution, which significantly speeds up the initial process of loading schemas and precomputing tableaux, without significantly slowing down the interactive process of adding and removing correspondences, is the *hybrid algorithm*. The main idea behind this algorithm is to precompute only a bounded number of source tableaux and target tableaux (up to a certain threshold, such as 100 tableaux in each schema). We give priority to the top tableaux (i.e., tableaux that include top-level set-type elements without including the more deeply nested set-type element). When a user interacts with the Clio GUI, she would usually start by browsing the schemas from the top and, hence, by adding top-level correspondences, that will match the precomputed skeletons.

However, correspondences between elements that are deeper in the schema trees may fail to match on any of the precomputed skeletons. We then generate, on the fly, the needed tableaux based on the endpoints of such a correspondence. Essentially, we generate a source tableau that includes all the set-type elements that are ancestors of the source element in the correspondence (and similarly for the target). The tableaux are then closed under the chase, thus including all the other schema elements that are associated via foreign keys. Next, the data structures holding the sub-tableaux and the sub-skeleton relationships are updated, incrementally, now that the tableaux and a new skeleton have been added. The new correspondence will then match the newly generated skeleton, and a new mapping can be generated. Overall, the performance of adding a correspondence takes a hit, but the amount of tableaux computation is limited locally (surrounding the end-points) and is usually quite acceptable. As a drawback, the algorithm may lose its *completeness*, in that there may be certain associations between schema elements that will no longer be considered (they would appear if we were to compute all the tableaux as in the basic algorithm). Still, this is a small price to pay, compared to the ability to load and map (at least partially) two complex schemas.

Performance evaluation: mapping MAGE-ML We now give an idea of the effectiveness of the hybrid generation algorithm by illustrating it on a mapping scenario that is close to worst case in practice. We load the same complex XML schema on both sides and experiment with the creation of the identity mapping. The schema that we consider is the MAGE-ML schema [5], intended to provide a standard for the representation of microarray expression data that would facilitate the exchange of microarray information between different data systems. MAGE-ML is a complex XML schema: it features many recursive types, contains 422 complex type definitions and 1515 elements and attributes, and is 172KB in size.

We perform two experiments. In the first one, we control the nesting level of the precomputed tableaux (maximum 6 nested levels of sets), but we set no limit on the total number of precomputed tableaux, when loading a schema. This experiment gives us a lower bound estimation on the amount of time and memory that the basic algorithm (that precomputes all the tableaux) requires. In the second experiment, we control the nesting level of the precomputed tableaux (also, maximum 6) *and* the total number of precomputed tableaux (maximum 110 per schema). This experiment shows the actual improvement that the hybrid algorithm achieves.

For the first experiment, we looked at the time to load the MAGE-ML schema on one side only (as source). This includes precomputing the tableaux as well as the sub-tableaux relationship. (The time to compile, in its entirety, the types of MAGE-ML, into the nested relational model poses no problem; it is less than 1 second.) We were able to precompute all (1030) the tableaux that obey the nesting level limit, in about 2.6 seconds. However, computing the sub-tableaux relationship (checking all pairs of the 1030 tableaux for the sub-tableau relationship) takes 74 seconds. The total amount of memory to hold the necessary data structures is 335MB. Finally, loading the MAGE-ML schema on the target side causes the system to run out of memory (512MB were allocated).

For the second experiment, we also start by loading the MAGE-ML schema on the source side. The precomputation of the tableaux (116 now) takes 0.5 seconds. Computing the sub-tableaux relationship (checking all pairs of the 116 tableaux to record if one is a subtableau of another) takes 0.7 seconds. The total amount of memory to hold the necessary data structures is 163MB. We were then able to load the MAGE-ML schema on the target side in time that is similar to that of loading the schema on the source side. The subsequent computation of the sub-skeleton relationship (checking all pairs of the 13456 = 116 x 116 skeletons to record whether one is a sub-skeleton of another) takes 35 seconds. The amount of memory needed to hold everything at this point is 251MB. We then measured the performance of adding correspondences. Adding a correspondence for which there are precomputed matching skeletons is 0.2 seconds. (This includes the time to match and the time to recompute the affected logical mappings.) Removing a correspondence requires less time. Adding a correspondence for which no precomputed skeleton matches and for which new tableaux and a new skeleton must be computed on the fly takes about 1.3 seconds. (This also includes the time to incrementally recompute the sub-tableaux and sub-skeleton relationships.) Overall, we found the performance of the hybrid algorithm to be quite acceptable and we were able to easily generate 30 of the (many!) logical mappings. Generating the executable script (query) that implements the logical mappings takes, additionally, a few seconds. To give a feel of the complexity of the MAGE-ML transformation, the executable script corresponding to the 30 logical mappings is 25KB (in XQuery) and 131KB (in XSLT).

The Clio implementation is in Java and the experiments were run on a 1600MHz Pentium processor with 1GB main memory.

3.2 Query Generation: Deep Union

There are two major drawbacks that the query generation algorithm described in Section 2.2 suffers from: there is no duplicate removal within and among query fragments, and there is no grouping of data among query fragments. For instance, suppose we are trying to create a list of orders with a list of items nested inside. Assume the input data comes from a simple relational table Orders (OrderID,ItemID). If the input data looks like $\{(o_1,i_1), (o_1,i_2)\}$, our nested query solution produces the following output data: $\{(o_1,(i_1, i_2)), (o_1,(i_1,i_2))\}$. The grouping of items within each order is what the user expected, but users may reasonably expect that only one instance of o_1 appears in the result. Even if we eliminate duplicates from the result of one query fragment, our mapping could result in multiple query fragments, each producing duplicates or extra information that needs to be merged with the result of another fragment. For example, assume that a second query fragment produces a tuple

[5] http://www.mged.org/Workgroups/MAGE/mage.html

```
WITH
 ExpSetFlat AS     -- Q1: The join and union of the relational data for ExpSet
 (SELECT DISTINCT
    x1.BIOLOGY_DESC AS exp_set_exp_set_header_biology_desc,
    x1.ANALYSIS_DESC AS exp_set_exp_set_header_analysis_desc,
    x0.ES_FK AS exp_set_id,
    …
    VARCHAR('Sk_GEML_2(' || x1.BIOLOGY_DESC || x1.ANALYSIS_DESC || … || ')') AS ClioSet0,
    VARCHAR('Sk_GEML_3(' || x1.BIOLOGY_DESC || x1.ANALYSIS_DESC || … || ')') AS ClioSet1
  FROM GENEX.EXPERIMENTFACTOR x0, GENEX.EXPERIMENTSET x1
  WHERE x0.ES_FK = x1.ES_PK
     UNION
  SELECT DISTINCT
    x1.BIOLOGY_DESC AS exp_set_exp_set_header_biology_desc,
    x1.ANALYSIS_DESC AS exp_set_exp_set_header_analysis_desc,
    x0.ES_FK AS exp_set_id,
    …
    VARCHAR('Sk_GEML_2(' || x1.BIOLOGY_DESC || x1.ANALYSIS_DESC || … || ')') AS ClioSet0,
    VARCHAR('Sk_GEML_3(' || x1.BIOLOGY_DESC || x1.ANALYSIS_DESC || … || ')') AS ClioSet1
  FROM GENEX.TREATMENTLEVEL x0, GENEX.EXPERIMENTSET x1
  WHERE x0.ES_FK = x1.ES_PK),
 ExpFactorFlat AS    -- Q2: The join of relational data for ExpFactor
 (SELECT DISTINCT
    VARCHAR('SK29(' || x0.FACTOR_NAME || … || ')') AS exp_factor_id,
    x0.FACTOR_NAME AS exp_factor_factor_name,
    …
    VARCHAR('Sk_GEML_2(' || x1.BIOLOGY_DESC || x1.ANALYSIS_DESC || … || ')') AS InSet
  FROM GENEX.EXPERIMENTFACTOR x0, GENEX.EXPERIMENTSET x1
  WHERE x0.ES_FK = x1.ES_PK),
 TreatmentLevelFlat AS   -- Q3: The join of relational data for TreatmentLevel
 (SELECT DISTINCT
    VARCHAR('SK109(' || x0.NAME || ',' … || ')') AS treatment_id,
    x0.NAME AS treatment_treatment_name,
    VARCHAR('Sk_GEML_4 (' || 'SK110(' || x0.NAME || x1.RELEASE_DATE || … || ')') AS ClioSet0,
    VARCHAR('Sk_GEML_3(' || x1.BIOLOGY_DESC || x1.ANALYSIS_DESC || … || ')') AS InSet
  FROM GENEX.TREATMENTLEVEL x0, GENEX.EXPERIMENTSET x1
  WHERE x0.ES_FK = x1.ES_PK),
```

Figure 6: SQL/XML script, relational part.

$\{(o_1,(i_3))\}$. We would expect this tuple to be merged with the previous result and produce only one tuple for o_1 with three items nested inside. We call this special union operation *deep-union*.

We illustrate the algorithm by showing the generated query in the case of SQL/XML, a recent industrial standard that extends SQL with XML construction capabilities. For the example, we assume that we are only interested in generating the transformation for two of our logical mappings: m_1 (mapping experiment sets with their associated experiment factors) and m_2 (which is similar to m_1 and maps experiment sets with their associated treatment levels).

The generated SQL/XML script can be separated in two parts. The first part (shown in Figure 6) generates a flat representation of the output in which a collection of tuples is represented by a system generated ID, and each tuple contains the ID of the collection it is supposed to belong to. The purpose of the second part (Figure 7) is to reconstruct the hierarchical structure of the target by joining tuples based on their IDs (i.e., joining parent collections with the corresponding children elements based on IDs). The final result is free of duplicates and merged according to the deep union semantics.

Briefly, Q1 joins `EXPERIMENTFACTOR` with `EXPERIMENTSET` and will be used to populate the atomic components at the `exp_set` level in the target (thus, not including the atomic data that goes under `exp_factor` and `treatment`). Two set-IDs are generated in Q1 (under the columns `ClioSet0` and `ClioSet1`), for each different set of values that populate the atomic components at the `exp_set` level. The first one, `ClioSet0`, will be used to group `exp_factor` elements under `exp_set`, while `ClioSet1` will be used to group `treatment` elements. The values for the set-IDs are generated as strings, by using two distinct Skolem functions that depend on all the atomic data at the `exp_set` level. The atomic data for `exp_factor` and `treatment` are created by Q2 and Q3, respectively. In both cases, a set-ID (named `InSet`) is created to capture what experiment set the data belongs to. The main idea here is that, as long as the values that go into the atomic components at the `exp_set` level are the same, the InSet set-ID will match the set-ID stored under `ClioSet0` (in the case of Q2) or `ClioSet1` (in the case of Q3). On a different note, we remark that all queries that appear in the first half of the script (e.g., Q1, Q2, and Q3) use the `DISTINCT` clause

```
 ExpFactorXML AS  -- Q4: Add XML tags to the data from Q2.
 (SELECT
    x0.InSet AS InSet,
    xml2clob(
      xmlelement(name "exp_factor",
        xmlattribute(name "factor_name", x0.exp_factor_factor_name),
        xmlelement(name "id", x0.exp_factor_id),
        …
      )) AS XML
  FROM ExpFactorFlat x0),

 TreatmentLevelXML AS  -- Q5: Add XML tags to the data from Q3.
 (SELECT
    x0.InSet AS InSet,
    xml2clob(
      xmlelement(name "treatment",
        xmlattribute(name "id", x0.treatment_id),
        xmlattribute(name "treatment_name", x0.treatment_treatment_name)
      )) AS XML
  FROM TreatmentLevelFlat x0),

-- Q6: Combines the results of Q1, Q4, and Q5 into one XML document.
SELECT xml2clob(xmlelement (name "exp_set",
  xmlattribute (name "id", x0.exp_set_id),
  …
  xmlelement (name "exp_set_header",
    xmlelement (name "biology_desc", x0.exp_set_exp_set_header_biology_desc),
    xmlelement (name "analysis_desc", x0.exp_set_exp_set_header_analysis_desc),
    xmlelement (name "exp_factors_list",
      (SELECT xmlagg (x1.XML)
       FROM ExpFactorXML x1
       WHERE x1.InSet = x0.ClioSet0)),
    xmlelement (name "treatment_list",
      (SELECT xmlagg (x1.XML)
       FROM TreatmentLevelXML x1
       WHERE x1.InSet = x0.ClioSet1)))
  )) AS XML
FROM ExpSetFlat x0
```

Figure 7: SQL/XML script continued, XML construction part.

to remove duplicate values.

In the second half of the script, the query fragments Q4 and Q5 perform the appropriate XML tagging of the results of Q2 and Q3. Finally, Q6 tags the `exp_set` result of Q1 and, additionally, joins with Q4 and Q5 using the created set-IDs, in order to nest all the corresponding `exp_factor` and `treatment` elements.

4. REMAINING CHALLENGES

There remain a number of open issues regarding scalability and expressiveness of mappings. Complex mappings sometimes need a more expressive correspondence selection mechanism than that supported by Clio. For instance, deciding which group of correspondences to use in a logical mapping may be based on a set of predicates. We are also exploring the need for logical mappings that nest other logical mappings inside. Finally, we are studying mapping adaptation issues that arise when source and target schemas change.

5. REFERENCES

[1] P. Bernstein. Applying Model Management to Classical Meta Data Problems. In *CIDR*, 2003.

[2] R. Fagin, P. Kolaitis, L. Popa, and W.-C. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. In *PODS*, 2004.

[3] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. In *ICDT*, 2003.

[4] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, 2002.

[5] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. Supporting Executable Mappings in Model Management. In *SIGMOD*, 2005.

[6] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema Mapping as Query Discovery. In *VLDB*, 2000.

[7] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *VLDB*, 2002.

[8] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4):334–350, 2001.

[9] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh, and V. Y. Lum. EXPRESS: A Data EXtraction, Processing, and REStructuring System. *TODS*, 2(2):134–174, 1977.