Simplifying Information Integration: Object-Based Flow-of-Mappings Framework for Integration

Bogdan Alexe[1], Michael Gubanov[2], Mauricio A. Hernández[3], Howard Ho[3], Jen-Wei Huang[4], Yannis Katsis[5], Lucian Popa[3], Barna Saha[6], and Ioana Stanoi[3]

[1] University of California, Santa Cruz
[2] University of Washington
[3] IBM Almaden Research Center
[4] National Taiwan University
[5] University of California, San Diego
[6] University of Maryland

Abstract. The Clio project at IBM Almaden investigates foundational aspects of data transformation, with particular emphasis on the design and execution of schema mappings. We now use Clio as part of a broader data-flow framework in which mappings are just one component. These data-flows express complex transformations between several source and target schemas and require multiple mappings to be specified. This paper describes research issues we have encountered as we try to create and run these *mapping-based data-flows*. In particular, we describe how we use *Unified Famous Objects* (UFOs), a schema abstraction similar to business objects, as our data model, how we reason about flows of mappings over UFOs, and how we create and deploy transformations into different runtime engines.

Key words: Schema Mappings, Schema Decomposition, Mapping Composition, Mapping Merge, Data Flows

1 Introduction

The problem of transforming data between different schemas has been the focus of a significant amount of work both in the industrial and in the research sector. This problem arises in many different contexts, such as in exchanging messages between independently created applications or integrating data from several heterogeneous data sources into a single global database.

Clio [14, 16, 9], a research prototype jointly developed by IBM Almaden and the University of Toronto, investigated algorithmic and foundational aspects of schema mappings. Figure 1 depicts Clio's architecture and demonstrates, at a high-level, the steps involved in transforming data structured under a schema S (called the *source schema*) to data structured under a different schema T

(called the *target schema*). First, the user specifies a set of correspondences between attributes of S and T using Clio's graphical user interface. Based on these attribute-to-attribute correspondences and the semantic constraints expressed by S and T, Clio generates a declarative specification of the data transformation. This declarative specification, which we call *schema mapping*, is formally expressed as a set of logical *source-to-target constraints* [8]. Finally, Clio translates the declarative schema mapping to executable code that performs the data transformation.

Clio contains several code generation modules that target different languages. For instance, when mapping xml-to-xml data, Clio can generate XQuery and XSLT scripts from the same mapping. Alternatively, when mapping relational data, Clio can generate SQL queries.

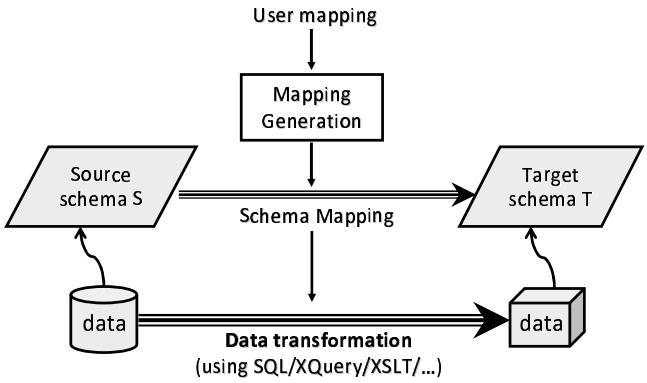

Fig. 1. Clio Architecture

Complex information integration tasks generally need the orchestration or flow of multiple data tranformation tasks. For instance, data transformations commonly contain many intermediate steps, such as shredding nested data into relations, performing joins, computing unions or eliminating duplicates. Parts of the flow are in charge of extracting relevant information from raw data sources, while other parts are in charge of transforming data into a common representation. Later parts of the flow are in charge of deploying the trasformed data into target data marts or reports. Single monolithic mappings, those that map from a source schema into a target schema, cannot capture these complex tranformation semantics. And even in cases when a mapping can capture the tranformation semantics, the source and target schemas might be large and complex, making it difficult for designers to create and maintain the mapping.

In this paper we discuss *Calliope*, a data flow system in development at IBM Almaden that uses mappings as a fundamental building block. Instead of designing a monolithic schema mapping, the users of *Calliope* create a flow of smaller, relatively simpler and easier to understand, mappings among small schemas. As

the name implies, the mappings are staged in a dependency graph and earlier mappings produce intermediate results that are used by later mappings in the flow (Figure 2). The individual mappings themselves can be designed using a mapping tool (e.g., such as Clio, which is now a component of *Calliope*). We believe the adoption of flows of mappings improves reusability, as commonly used transformation components can be reused either within the same transformation task (e.g. if an address construct appears twice within the same source schema) or across transformation tasks.

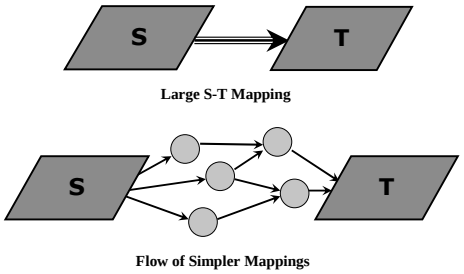

Fig. 2. Monolithic mappings (traditional approach) and flow of mappings in *Calliope*

However, representing transformations as flows of mappings creates a number of interesting technical challenges. First, we need a uniform data model that represents data flowing over the graph of mappings. Second, to better understand the transformation semantics of a flow of mappings, we need to compose and *merge* mappings before generating run-time objects. Last, we need to generate transformation scripts over a combination of different runtime systems, each with different data transformation capabilities. We now briefly discuss each of these challenges in the rest of this section.

Unified Famous Objects (UFOs). One of the goals of *Calliope* is to raise the level of abstraction from schemas, and bring it one step closer to business objects. This is accomplished through the use of *Unified Famous Objects (UFOs)*. A UFO is a unified representation of a high-level concept, such as employee or address. One or more of these concepts may be embedded in a schema. Instead of mapping directly from an arbitrary complex source schema to an arbitrary complex target schema, which requires understanding their exact relationship, the mapping designer can focus on a single schema fragment at a time. Each such fragment is mapped to one or more UFOs that capture the concepts represented in the current schema fragment. The specification of the end-to-end transformation is then assembled from the smaller mappings between the source schema and the UFOs, the mappings between the UFOs and the target schema as well as from mappings between the UFOs themselves. We believe this approach facilitates thinking in terms of high-level concepts, which are closer to what users

understand. UFOs also allow mapping reuse. Mappings between UFOs can be reused in various contexts where transformations are needed for the concepts they represent. Finally, this approach is modular: When changes occur over the schemas, or the semantics of the desired transformation changes, only the relevant parts of the smaller mappings involving UFOs need to be updated. In this new approach the mapping designer will be able to focus on the mappings between the end schemas and the UFOs, and reuse as much as possible previously designed mappings between UFOs.

Mapping Merge. To allow the execution of flows of mappings, *Calliope* employs novel mapping technology that allows for the automatic assembly of initially uncorrelated mappings into larger and richer mappings. This technology relies on two important mechanisms: mapping merging and mapping composition. *Mapping merging*, or correlation, is responsible for joining and fusing data coming from initially uncorrelated mappings. *Mapping composition* [7, 2] on the other hand, allows assembling sequences of mappings into larger mappings. Intuitively, mapping merging combines mappings that correspond to parallel branches in the mapping flow graph, while mapping composition combines mappings that correspond to sequential paths. We note that mapping merging, which is discussed in some level of detail in Section 4, is an operator on schema mappings that has been largely undeveloped until now.

Unified Flow Model (UFM) Framework. Some of our use cases involve users that want to design a single transformation using a *combination* of many different data transformation tools, such as mapping tools, ETL tools or query languages [3]. Such cases are becoming increasingly common in practice for two main reasons: First, some data transformation tools are not expressive enough to represent a data transformation and thus the transformation generated by the tool has to be augmented with additional operators. For instance, Clio does not allow sorting of results. Therefore if sorting is desirable, the transformation generated by Clio has to be augmented with sorting in a language that offers the appropriate construct, such as SQL. Second, even when a single data transformation tool can express the entire transformation, users familiar with different tools want to collaborate on the design of the same transformation. For example, analysts designing the coarse outline of a transformation in Clio would like to have it extended with lower-level details by programmers, who are familiar with ETL-tools.

To address these interoperability requirements we designed the *Unified Flow Model (UFM) framework*. At the heart of this framework lies the Unified Flow Model (UFM), which allows the representation of a data transformation in a tool-independent way. Given the UFM framework, all it takes to make *Calliope* interoperate with other data transformation tools is to design procedures that translate the internal representation of any data transformation tool to UFM and vice versa. The main challenges in the context of this framework is finding the right language for UFM and designing the translations between UFM and

the internal languages for the various data trasformation tools.

The following sections describe in more details the main components of *Calliope*. Section 2 introduces the notion of UFOs and describes the UFO repository. Section 3 describes how to decompose a schema into a set of UFOs. Section 4 gives an illustration of the mapping merging technique used in *Calliope*, and Section 5 presents the UFM framework.

We note that *Calliope* is at an early stage and many of the ideas presented in this paper are still under development.

2 Unified Famous Objects (UFOs)

Traditional mapping tools allow the specification of a transformation by defining a mapping from a source schema to a target schema. However this approach becomes problematic as the schemas become larger and the transformation more complex. Large schemas are hard to understand and it is even harder to create a mapping between them in a single step. To remedy this problem, *Calliope* allows users to split the large and complex mapping of the source to the target schema into more easily comprehensible steps (which are themselves composed of multiple mappings) that are based on the use of intermediate Unified Famous Objects (UFOs).

A UFO, similar to a business object, is a flat object representing a simple concept, such as an employee, a product or an article. Being similar to a business object, it is a higher-level abstraction than a schema and, as such, closer to the understanding of the mapping designer. In *Calliope*, UFOs can be either defined manually or extracted automatically from a source that provides standardized schemas, such as Freebase or OAGI. Once created, they are stored in the metadata repository of the system, ready to be used in mappings. To model semantic relationships between related UFOs, the metadata repository can also hold mappings between UFOs.

Given the metadata repository, mapping a source schema to a target schema translates to the following steps: First, the designer finds the UFOs in the metadata repository that are relevant to the source schema. To facilitate this process, the metadata repository offers an interface that allows the designer to browse and query its contents. Once the relevant UFOs are found, the designer creates a separate mapping between the source schema fragment representing a particular concept and the corresponding UFO. After finishing this process for the source schema, the designer repeats the symmetrical procedure for the target schema: find UFOs that are relevant to the target schema and design mappings from each of them to the target schema. The resulting end-to-end transformation between the source schema and the target schema is then the flow of mappings composed of: a) mappings from the source schema to UFOs, b) mappings from UFOs to the target schema and c) any number of intermediate mappings that may be needed between the UFOs themselves. Some of the intermediate UFO-to-UFO

mappings may have to be designed at this point, but some could be reused from the metadata repository (if they already exist).

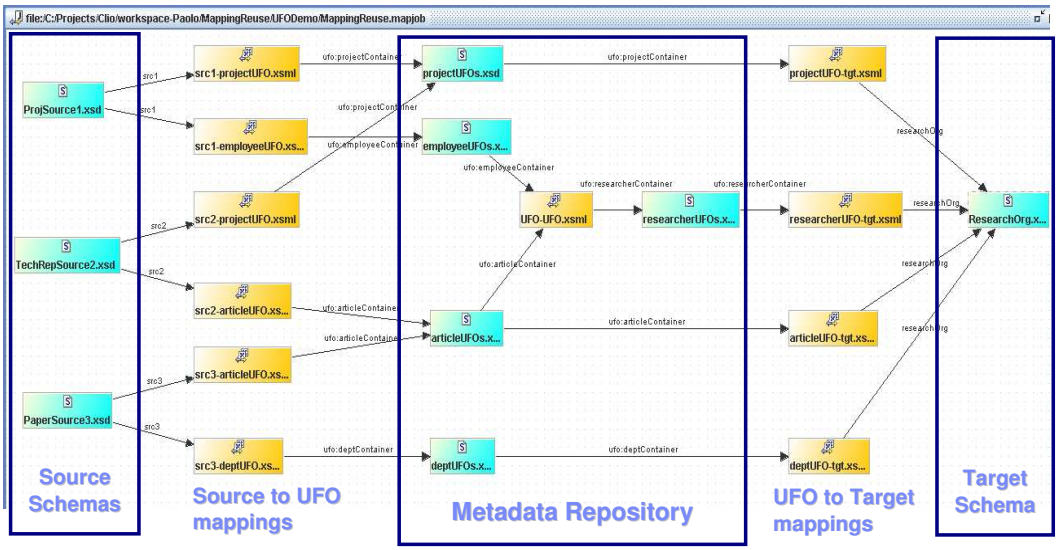

Fig. 3. Flows of Schema Mappings

Figure 3 shows a sample flow of mappings between source and target schemas in the presence of UFOs. The picture displays two types of nodes: schema nodes (used to represent both source/target schemas and UFOs) and mapping nodes. Source schemas (inside the box on the left) are mapped to UFOs (inside the box in the middle). For instance, the source on the top contains information on projects and employees and therefore it has been mapped to the corresponding two UFOs, representing projects and employees, respectively. Apart from the mappings from source schemas to UFOs, the picture also shows mappings between UFOs (inside the middle box). For example, the UFO representing a researcher contains both employee and article information and therefore it has mappings from the corresponding UFOs. Finally, UFOs are mapped to the target schema (shown inside the box on the right).

Using UFOs in the mapping process yields several advantages: First, it allows the designer to decompose a large source-to-target mapping into many smaller mappings that involve the UFOs. Since each individual mapping is relatively small, it is easy to create and maintain. Second, by storing in the repository the mappings between schemas and UFOs, we can improve the precision of matching algorithms by learning from previous mappings. For instance, we can store attribute name synonyms next to each UFO attribute to help with subsequent matching. More importantly, UFOs allow for standardization and reuse. A large part of the mapping and transformation logic can be expressed in terms of a

fixed set of UFOs describing a domain, and this logic can then be reused and instantiated in different applications (on the same domain).

The presence of the UFO repository also creates some important challenges. First, as the number of UFOs increases it becomes increasingly harder to find the UFOs that are relevant to a given schema. To remedy this problem, *Calliope* contains a schema decomposition algorithm, described in the following section. Second, the mappings that are created between the source and target schemas and the UFOs, as well as the mappings that relate UFOs and are potentially extracted from the repository, are initially uncorrelated, and possibly independently designed. The main challenge here is to orchestrate the flow of uncorrelated mappings into a global mapping that describes a meaningful end-to-end transformation. As a solution to this problem, *Calliope* relies on a mapping merge mechanism. We give an overview of this mechanism in Section 4.

3 Schema Decomposition

Today a user will handle new schemas integration tasks by designing the transformation operator flow manually, from scratch. Since defining these operators is cumbersome [14, 16], it is important to bootstrap the process with relevant operator flows. At the core of reusing operator flows is the task of recognizing commonalities between a new input schema and fragments of previously used schemas. Thus, an essential step in using UFOs for schema mapping is the *decomposition* of the source and target scheams into the right set of UFOs. *Schema decomposition* is the technique of automatically selecting a collection of concepts from a given repository, which together form a good coverage of a new input schema. In general, schema decomposition enables the understanding and representation of large schemas in terms of the granular concepts they represent. This step should automatically propose good decompositions, and allow the user to further modify and enhance them. Schema decomposition should be both efficient in tranversing a large space of UFOs, and effective in producing decompositions of high quality.

Consider for example the schema in Figure 4(i). An integration advisor, using the techniques of schema decomposition can automatically identify that the same *Address* format has been used before. It will then propose transformation flows for this specific format of *Address*.

There has been a lot of work in semi-automated schema matching, proposing solutions that are based on schema and instance level, element and structure-level, language and constraint-based matching, as well as composite approaches [17, 18]. Some of the different systems that have been developed for matching include SemInt [10], CUPID [12], SF [13], LSD [5], GLUE [6], COMA [4], TranScm [15], Momis [1] etc. The reuse of previously determined match results proposed by Rahm and Bernstein in [17] has also attracted recent research focus and has been successful in significantly reducing the manual match work [4, 11]. However the reuse of matching is limited to element to element match and not on any larger matching concepts. Most of these works can only handle small and

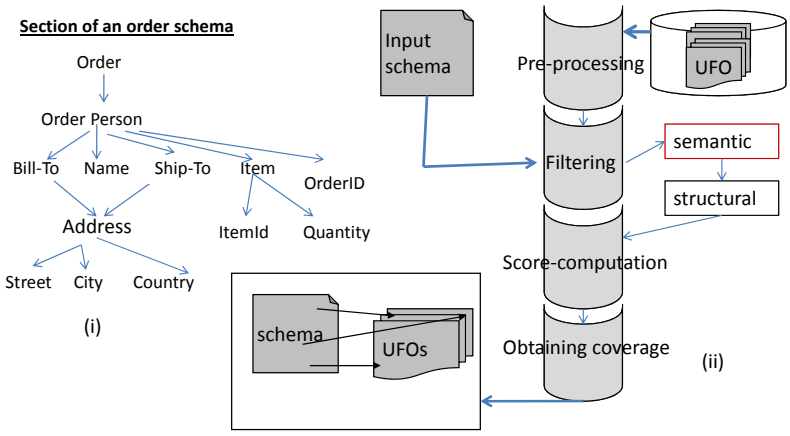

Fig. 4. (i) A Schema-Graph, (ii) Components Used in Schema Decomposition

structurally simple schemas and define direct schema matching from source to target. In contrast, our proposed schema decomposition technique can be used to exploit reuse on much larger matching concepts (*UFO*s) and make the task of schema-matching and mapping between structurally complex schemas substantially easier.

Enabling reuse goes beyond the design of schema mappings. It can be exploited by ETL tools, Mashups and in general by Information Integration systems. Consider a schema for a new Mashup feed, where similar fragments of the schema have been used in other data sources. Then the automated advisor can detect similarities, and advise the user of opportunities for exploiting the new data in interesting ways. In Information Integration, let us consider two schemas that are understood in terms of the concepts from the repository. Then common concepts point to schema matches, and can be used to efficiently direct users through the task of integrating the two schemas. In all these aspects, schema decomposition will serve as the fundamental tool.

3.1 An overview of the technique

Given a source and a target schema, and a repository of *UFO*s, schema decomposition aims towards "covering" the schemas with UFOs as well as possible. Known matching and transformations between UFOs can then be exploited to obtain a source to target schema-mapping. Populating the repository with the right schema fragments can be thought of as a preprocessing step. By contrast, schema decomposition needs to be handled online, as new input schemas are introduced. Let us denote an input schema by S, and the repository by $\mathcal{R}$. Schema decomposition identifies the matching concepts (schema fragments) U_i's from $\mathcal{R}$ that best cover S. The selection of the U_i's and their corresponding positioning

to cover S are factors in the quality of the coverage. For efficiency, we use a filter/evaluate/refine approach as described below.

The repository of schema fragments U_i can be quite large, and most of these fragments will not match at all S. Therefore, it is necessary to reduce the search space and select only fragments from the repository that have potential to be relevant. This filtering is performed in two phases. In the first phase, a semantic filter is used that compares labels of each U_i against labels in S. Note that the structural relationship between labels is not yet taken into account. The output of the semantic filter is the set of candidates U_i that have any semantic labels in common with S. However, this does not test for the structural similarity. That is, do the labels in any of the U_i's match a cohesive region of S? And if yes, how much structural flexibility do we allow when considering a match? The structural filter evaluates these questions and further refines the candidate set.

The next step in schema decomposition involves a score computation model, in order to compute the quality scores for the U_i's in the candidate set. The score for each U_i is computed based on the portions of S that U_i can potentially cover. Note that considering all possible sub-regions of S and calculating quality scores of U_i's for each region is an exponential process. In practice the score computation model needs to rely on a quality measure that avoids this exponential running time and can be computed efficiently.

The last step is an optimization step that considers all the selected *UFO*s together with their quality score and 1) selects a subset of *UFO*s and 2) positions them relative to S so that the aggregated quality score is maximized. Figure 4(ii) illustrates this framework. This last step is NP-Complete and thus an exhaustive approach will take exponential time and is potentially of no practical interest. Instead, approximation algorithms that are fast and guarantee near-optimality of the result quality should be used.

4 Orchestrating Flows of Schema Mappings

As mentioned before, the mappings between the end schemas and the UFOs, as well as the mappings that relate UFOs extracted from the repository, are initially uncorrelated. To express a meaningful end-to-end transformation, the flow of independently designed, uncorrelated mappings needs to be assembled into a global mapping. To achieve this, *Calliope* relies on a mechanism for merging smaller, uncorrelated mappings into larger, richer mappings, through joining and fusing data coming from individual mappings.

We give a brief overview of the merging technique through the use of a simple example. Consider the scenario in Figure 5. It consists of a nested source schema representing information about projects and the employees associated with each project. Potentially following the schema decomposition phase, a set of three UFOs have been identified as relevant to the source schema: Project, Dept, and Employee. These are possibly extracted from the UFO repository and are standardized representations of the business objects project, department, and employee, respectively. Using schema matching techniques and the mapping

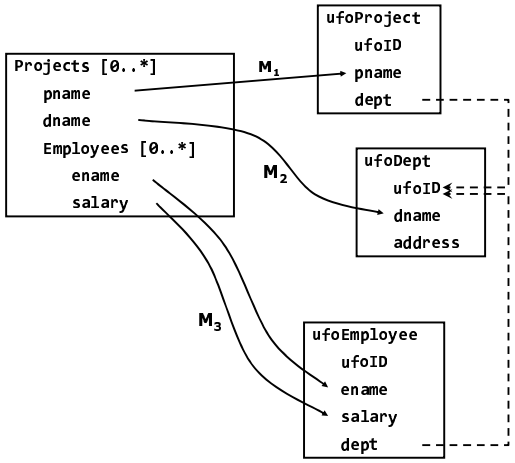

Fig. 5. Mapping Merge Scenario

design capabilities in Clio, three initial mappings M_1, M_2, M_3 are constructed between the source schema and each of the three UFOs. These mappings are decorrelated and, in a real life scenario, may be constructed by independent mapping designers. This mapping scenario may be part of a larger flow of mappings from the source schema, through the UFOs above, and possibly others, to one or more target schemas. The mappings in the scenario above can be expressed in a logical formalism similar to source-to-target constraints [8], as below:

$$M_1 : \text{Projects}(\text{p}, \text{d}, \text{E}) \rightarrow \text{ufoProject}(\text{PID}, \text{p}, \text{DID})$$
$$M_2 : \text{Projects}(\text{p}, \text{d}, \text{E}) \rightarrow \text{ufoDept}(\text{DID}', \text{d}, \text{A})$$
$$M_3 : \text{Projects}(\text{p}, \text{d}, \text{E}) \wedge \text{E}(\text{e}, \text{s}) \rightarrow \text{ufoEmployee}(\text{EID}, \text{e}, \text{s}, \text{DID}'')$$

For instance, mapping M_3 above states that for each project record in the source, and each employee record in the set of employees associated with that project, there must exist a `ufoEmployee` object where the values for the `ename` and `salary` attributes come from the corresponding attributes in the source employee record. However, according to M_3, the identifier of the `ufoEmployee` object as well as the value of the `dept` attribute remain unspecified, and are allowed to take some arbitrary values. Similarly, mappings M_1 and M_2 put in correspondence source project records to `ufoProject` and `ufoDept` objects, respectively. Note that according to the mappings above, the values of the `dept` attribute in `ufoProject` and `ufoEmployee` and the identifier of `ufoDept` are not correlated.

The mapping merge mechanism in *Calliope* rewrites and combines the initial mappings to generate a set of richer mappings that support a meaningful transformation where target data records the "correct" correlations. As a first

step, the merging technique takes advantage of any constraints that may be present among the UFOs. In the example above, there are two referential constraints, indicated through dashed lines in figure 5. These constraints indicate that the `dept` attributes in `ufoProject` and `ufoEmployee` must be identifiers for a `ufoDept`. The merge mechanism considers these constraints and rewrites the initial mappings into the following mappings:

$$\begin{aligned} M_1' : \text{Projects}(\text{p}, \text{d}, \text{E}) &\rightarrow \text{ufoProject}(\text{PID}, \text{p}, \text{DID}) \\ &\quad \wedge \text{ufoDept}(\text{DID}, \text{d}, \text{A}') \\ M_3' : \text{Projects}(\text{p}, \text{d}, \text{E}) \wedge \text{E}(\text{e}, \text{s}) &\rightarrow \text{ufoEmployee}(\text{EID}, \text{e}, \text{s}, \text{DID}'') \\ &\quad \wedge \text{ufoDept}(\text{DID}'', \text{d}, \text{A}'') \end{aligned}$$

In the new set of mappings, M_1' is the result of merging M_1 and M_2, and M_3' is the result of merging M_3 and M_2. Note that in M_1', the `ufoProject` and `ufoDept` are correlated via the department identifier field. A similar correlation exists in M_3' between `ufoEmployee` and `ufoDept`. Note that there is no correlation (yet) between M_1' and M_3'.

An additional merging step applies by taking advantage of the fact that M_3' is a "specialization" of M_1': the latter mapping defines behavior for project records, in general, whereas the former considers, additionally, the employee records associated to the project records. Concretely, we can merge the more specific part of M_3' (i.e., the employee mapping behavior) into M_1' as a nested sub-mapping. The resulting mapping is the following:

$$\begin{aligned} \text{Projects}(\text{p}, \text{d}, \text{E}) &\rightarrow \text{ufoProject}(\text{PID}, \text{p}, \text{DID}) \\ &\quad \wedge \text{ufoDept}(\text{DID}, \text{d}, \text{A}') \\ &\quad \wedge [\ E(e, s) \rightarrow \text{ufoEmployee}(\text{EID}, \text{e}, \text{s}, \text{DID})\ ] \end{aligned}$$

The resulting overall mapping includes now all the "right" correlations between projects, departments and employees. The top-level mapping transforms all project records, by creating two correlated instances of `ufoProject` and `ufoDept` for each project record; the sub-mapping, additionally, maps all the employee records that are associated with a project, and creates instances of `ufoEmployee` that are all correlated to the same instance of `ufoDept`.

This simple example gives just an illustration of the techniques for merging mappings that are used in *Calliope*. The full details of the complete algorithm for merging will be given elsewhere.

5 Unified Flow Model Framework

As a schema mapping tool, Clio aimed in allowing people to transform data between different schemas. However, it is far from being the only tool available for this task. Data transformations can nowadays be carried out through a variety

	Mapping Tools	*ETL Tools*	*DBMS*
Design Paradigm	Mappings	Flow of ETL Operators	Query
Target Audience	Analysts	Programmers	DB Experts
Execution Engine	XQuery engine, XSLT engine etc.	Custom	Query Engine

Table 1. Properties of Mapping Tools, ETL Tools and DBMSs

of different tools, such as mapping tools (e.g. Clio, Stylus Studio), Extract-Transform-Load (ETL) tools (e.g. Datastage), Database Management Systems (DBMSs) (e.g. DB2, SQL Server, Oracle) etc.

These tools offer different *paradigms to design* the transformation (and thus they have different target audiences) and they employ different *engines to execute* the designed transformation. Table 1 summarizes the design and runtime components for mapping tools, ETL tools and DBMSs. For instance, mapping tools allow the declarative design of transformations (through mappings) and therefore they are best suited to analysts who want to design a transformation at a high level. In contrast, ETL tools have a more procedural flavor, allowing the design of a transformation through the composition of a set of primitive ETL operators. This makes them the ideal platform for highly-skilled programmers who want to design complex workflows. Similarly, mapping tools and ETL tools employ different engines to execute the transformation. Mapping tools usually translate the transformation down to a query language like XQuery or XSLT, while ETL tools often contain their custom execution engines optimized for the supported set of operators.

However by tying together a certain design component and runtime component, existing tools severely restrict the users in two significant ways: First, they force the users to employ both the design and runtime component supported by the tool. For example, a user cannot design a transformation in Clio and execute it using the highly optimized internal engine of an ETL tool, since the ETL's runtime engine can only execute transformations designed through the ETL paradigm. Second, they do not allow users to employ different design paradigms to design a single flow. For example, an ETL programmer willing to extend the transformation sketched by an analyst in Clio, has to start from scratch as there is no automatic provision for the translation between design paradigms supported by different tools.

To alleviate these problems, *Calliope* is based on the *Unified Flow Model Framework*, shown in Figure 6. The heart of this framework is the *Unified Flow Model* (UFM); a model that represents data transformations in a tool-independent way. Different design components can be added in the system in a modular way by specifying a procedure for translating the internal representation of the design component to UFM and vice versa. Similarly, a developer can add an execution engine to the framework by specifying how a transfor-

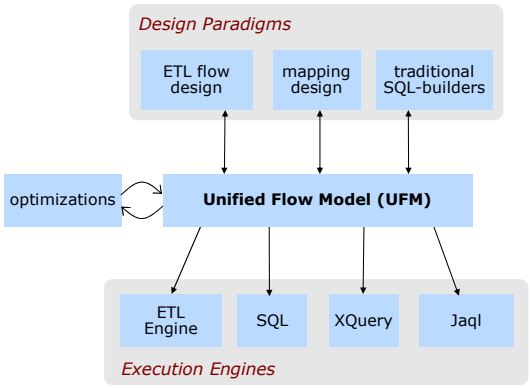

Fig. 6. Unified Flow Model Framework

mation described in UFM can be translated to a language recognized by the particular execution engine. The result is a framework, where users can design a data transformation in one or more design components (originating from diverse tools, such as mapping tools, ETL tools etc.) and subsequently execute it on any execution engine that has been added to the framework.

The implementation of the UFM framework poses many important challenges caused by the differences between various design components (respectively execution engines). First of all, the translation between different design paradigms might not be always possible, since they have in general different expressive power. Should such translations fail or should the system try to translate the largest subset of the transformation possible? Second, although a transformation can be optimized at the UFM level, there will be some optimizations that are execution engine dependent. What types of optimizations can be done on the UFM level and which optimizations require knowledge of the execution engine on which the transformation is going to be ran? These are a few of the questions that arise in the implementation of the UFM framework.

As a first step towards our vision, we have designed the following components of the UFM framework: a) the UFM model as a flow of ETL-like operators that can express most common data transformations (which contain all transformations currently supported by Clio's mapping language), b) the translation from Clio's mapping language to UFM and c) the translation from UFM to Jaql; a JSON query language that contains a rewriting component that translates queries to MapReduce jobs that can be executed in Hadoop. This corresponds to the addition of two components in the architecture shown in Figure 6: the mapping design component (with a unidirectional arrow into UFM) and the Hadoop execution engine.

6 Conclusion

We have discussed the main components of *Calliope*, a system for creating and maintaining flows of mappings. Our main motivation for *Calliope* was to extend and reuse the basic schema mapping operations explored in Clio in more complex and modular data transformation jobs.

We believe that complex mappings are difficult to build and manage in one step. Not only is it conceptually hard to understand the relationships between schemas with a potentially very large number of elements but it is also hard to visualize and debug them. *Calliope* allows users to express mappings in terms of higher-level objects, such as business objects, that are easier to understand and do not couple implementation with semantics. Further, mappings in *Calliope* are smaller and modular, increasing the opportunity for reuse.

Acknowledgements. We acknowledge Hamid Pirahesh for his original suggestion to use UFOs and also for his continuous feedback on this work.

References

1. Sonia Bergamaschi, Silvana Castano, Maurizio Vincini, and Domenico Beneventano. Semantic integration of heterogeneous information sources. *Data Knowl. Eng.*, 36(3):215–249, 2001.
2. Philip A. Bernstein, Todd J. Green, Sergey Melnik, and Alan Nash. Implementing Mapping Composition. In *Proceedings of VLDB*, pages 55–66, 2006.
3. Stefan Dessloch, Mauricio A. Hernández, Ryan Wisnesky, Ahmed Radwan, and Jindan Zhou. Orchid: Integrating Schema Mapping and ETL. In *ICDE*, pages 1307–1316, 2008.
4. Hong-Hai Do and Erhard Rahm. Coma: a system for flexible combination of schema matching approaches. In *VLDB '02*, pages 610–621, 2002.
5. AnHai Doan, Pedro Domingos, and Alon Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *SIGMOD '01*, pages 509–520, 2001.
6. Anhai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. Learning to map between ontologies on the semantic web. In *WWW '02*, pages 662–673, 2002.
7. Ronald Fagin, Phokion Kolaitis, Lucian Popa, and Wang-Chiew Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. In *PODS*, pages 83–94, 2004.
8. Ronald Fagin, Phokion G. Kolaitis, Rene J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89 – 124, 2005.
9. Ariel Fuxman, Mauricio A. Hernández, Howard Ho, Renée J. Miller, Paolo Papotti, and Lucian Popa. Nested Mappings: Schema Mapping Reloaded. In *Proceedings of VLDB*, pages 67–78, 2006.
10. Wen-Syan Li and Chris Clifton. Semint: a tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data Knowl. Eng.*, 33(1):49–84, 2000.
11. Jayant Madhavan, Philip A. Bernstein, AnHai Doan, and Alon Halevy. Corpus-based schema matching. In *ICDE '05*, pages 57–68, 2005.

12. Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *VLDB '01*, pages 49–58, 2001.
13. Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm. In *ICDE '02*, pages 117–128, 2002.
14. Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *VLDB '00*, pages 77–88, 2000.
15. Tova Milo and Sagit Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB '98*, pages 122–133, 1998.
16. Lucian Popa, Yannis Velegrakis, Mauricio A. Hernández, Renée J. Miller, and Ronald Fagin. Translating web data. In *VLDB '02*, pages 598–609, 2002.
17. Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.
18. Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. *J. Data Semantics IV*, 3730:146–171, 2005.