ABSTRACT
We present GROM, a tool conceived to handle high-level schema mappings between semantic descriptions of a source and a target database. GROM rewrites mappings between the virtual, view-based semantic schemas, in terms of mappings between the two physical databases, and then executes them. The system serves the purpose of teaching two main lessons. First, designing mappings among higher-level descriptions is often simpler than working with the original schemas. Second, as soon as the view-definition language becomes more expressive, to handle, for example, negation, the mapping problem becomes extremely challenging from the technical viewpoint, so that one needs to find a proper trade-off between expressiveness and scalability.

1. INTRODUCTION
Many applications benefit from the availability of a semantic schema over a database, i.e., a set of views over the base tables that provide a richer description of the semantic relationships among the underlying data and a more accurate definition of the constraints. The use of such semantic views has been thoroughly studied for the purpose of query languages [6], data integration [1], and data access [2], but there are little studies of how the presence of these views impacts data exchange [4] applications.

Data exchange consists of moving data from a source database to a target database. This task is usually performed by developing schema mappings, i.e. executable transformations that specify how an instance of the source repository can be translated into an instance of the target.

In this paper, we present GROM [9, 8], a system conceived to support the management of mappings among view schemas. GROM was designed to handle mapping scenarios in which a semantic description is available over the target database, and possibly over the source database. It allows data architects to develop mappings among the two semantic schemas, rather than the underlying database schemas. Studying this variant of the problem is important for several reasons:

(i) The semantic web has increased the number of data sources on top of which such descriptions are developed.
(ii) Views play a key role in information integration since they are used to give clients a global conceptual view of the underlying data, which may come from external, independent and heterogeneous information systems [7].
(iii) Many of the base transactional repositories used in complex organizations often undergo modifications during the years, and may lose their original design. It is important to be able to run the existing mappings against a view over the new schema that does not change, thus keeping these modifications of the sources transparent to the users.

More generally, semantic schemas help to improve the overall design of the original schemas, and emphasize important semantic relationships and constraints that would not be apparent otherwise. Therefore, designing rich, high-level mappings between these schemas has often significant advantages. However, semantic schemas are virtual and mappings between them are not directly executable.

GROM solves the important problem of making these high-level mappings executable over the original database instances. It rewrites mappings between the two virtual semantic schemas under the form of standard mappings over the underlying concrete databases, in order to execute them and generate an instance of the target database from an instance of the source database. Under appropriate hypothesis, discussed in the next sections, the whole process happens in a completely transparent way, thus greatly simplifying the overall data-translation task.

Essential to this problem is the trade-off between expressiveness and complexity. In fact, the rewriting is fairly straightforward if views are conjunctive queries – it reduces to the standard view unfolding algorithm. However, conjunctive queries have a limited expressive power, unable to capture many semantic relationships between the data. Negation, for instance, is crucial to capture disjointness constraints and many classification rules.

The main concern behind the design of GROM was to provide an expressive view language that can truly benefit data architects in defining rich semantic abstractions. To this end, we adopt the language of non-recursive Datalog with negation. This makes the rewriting significantly more complex, as we discuss in Section 3.

In the following we describe how we plan to organize the demonstration of GROM. We outline the kind of mapping scenarios that will be considered with the help of a running example introducing the main features of the system. Given
the focus of this proposal, we have chosen to omit many of the technical details that are in published papers [9]. We concentrate on a description of the system from the user perspective and illustrate the main technical challenges and what an attendee may learn by playing with it.

The system is available under an open-source license at the following URL: http://db.unibas.it/projects/grom/.

2. MAPPING REWRITING

Assume we have the two relational schemas below and we need to translate data from the source to the target.

Source schema:
- S-Product(id, name, store, rating)
- S-Store(name, location)

Target schema:
- T-Product(id, name, store)
- T-Store(id, name, address, phone)
- T-Rating(id, product, thumbsUp)

Both schemas refer to the same domain, which includes data about products, stores, and ratings. Due to the different organization of the two databases, it is not evident how to define the source-to-target mapping. In particular, it is difficult to relate tuples in the T-Rating target table to those in the source. Suppose now that a richer semantic schema has been defined over the target, as shown in Figure 1. To simplify things, in this example we consider only that the target database comes with an associated semantic schema; we discuss the more general case in the next section.

The important observation here is that in many cases semantic concepts are closer to source data than physical target tables, and therefore the task of defining mappings is considerably simplified. In our example, notice how the views hide table T-Rating. As a consequence, the classification of a product in the target semantic schema is easily derived from ratings in the source database as follows: products with ratings consistently above 4 stars (out of 5) are the popular ones, those always graded less than 2 are considered to be unpopular, and the rest are average.

As is common [4], we use tuple generating dependencies (tgds) and equality-generating dependencies (egds) to express the mapping. In our case, the source-to-semantic translation can be expressed by using the following tgds with comparison atoms:

\[ m_0 : \forall pid, name, store, rating : \]
\[ S-Product(pid, name, store, rating) \rightarrow UnpopularProduct(pid, name) \]

\[ m_1 : \forall pid, name, store, rating : \]
\[ S-Product(pid, name, store, rating), rating \geq 2 \rightarrow AvgProduct(pid, name) \]

\[ m_2 : \forall pid, name, store, rating : \]
\[ S-Product(pid, name, store, rating), rating > 4 \rightarrow PopularProduct(pid, name) \]

\[ m_3 : \forall pid, name, store, rating, location : \]
\[ S-Product(pid, name, store, rating), S-Store(store, location) \rightarrow SoldAt(pid, sid, Store(sid, store, location)) \]

Intuitively, tgd \( m_0 \) specifies that, for each tuple in S-Product such that the value of rating is lower than 2, there should be an UnpopularProduct in the semantic schema. Similarly for \( m_1 \) and \( m_2 \). Mapping \( m_3 \) relates products and stores in the source to instances of SoldAt association in the semantic schema.

The mapping designer can also express a number of constraints about the target semantic schema under the form of egds.\(^1\) The egd below corresponds to a key constraint on PopularProduct: it states that whenever two popular products have the same name, their id must also be the same:

\[ e_0 : \forall id_1, id_2, n : \text{PopularProduct}(id_1, n), \text{PopularProduct}(id_2, n) \rightarrow id_1 = id_2 \]

In addition to being more natural, designing mappings over semantic schemas has another important benefit to the data architect. By taking advantage of the semantics of the views, the mapping designer does not need to care about the physical structure of the data in the target schema. As an example, s/he does not need to explicitly state in \( m_0 \), \( m_1 \), \( m_2 \) that popular, average, and unpopular products are also products. The class-subclass relationships are encoded within the view definitions, and we expect their semantics to carry on into the mappings.

This, however, is true provided that we are able to translate such a source-to-semantic virtual mapping into a classical, executable source-to-target mapping among the two physical databases. This is the main task performed by GROM, as discussed in the following section.

3. OVERVIEW OF THE SYSTEM

The main technical problem addressed by GROM, depicted in Figure 2, can be stated as follows. Assume we are given:

\(^1\)Previous papers [9] discuss how to handle foreign-key constraints as well.
The demonstration will illustrate what are the main challenges in handling semantic mappings and how the system

does it.

4. EXPERIENCES WITH THE SYSTEM

The demonstration will illustrate what are the main challenges in handling semantic mappings and how the system
solves them. Attendees will be able to interact directly with the system, in such a way that the process will resemble a hands-on tutorial. Following are the main lessons that can be learned from these experiences.

**Semantic Mappings Work!** Our experiences tell us that in many cases the availability of a view over the data may greatly simplify the mapping process. In these cases, data architects may greatly benefit from a tool like GROM.

One of the typical patterns is the one discussed in our running example: one of the data sources somehow rates source objects, and the mapping application requires to classify objects in the target based on these ratings, for example for the purpose of showing them to users under the form of web pages. Often, target relational schemas are not designed to properly address this kind of need. Being able to design a view over the target database that more closely reflects such an application requirement is a great asset in these scenarios.

Another, typical case, is the one of databases that come with poor designs, or lack integrity constraints. It is very difficult in these cases to design proper mappings. On the contrary, a clean-up view over the underlying databases may simplify things.

We intend to challenge the audience with different schemas and mapping scenarios. We will ask attendees to design mappings first using the original, relational schemas, and then over properly designed views, to let them grasp the real advantage of this approach.

**Semantic Mappings are Expensive!** At the same time, it is important to let users understand the concrete trade-off between having a flexible and expressive view-definition language, and the cost of executing the mappings.

As we mentioned, rewriting and executing the mappings is quite straightforward as soon as conjunctive queries are used as a view definition language. This, however, is not sufficient to capture the actual modeling requirements in many cases.

Negation is a powerful addition, but it comes at a cost. The full power of negation generates output mappings that include ded, so that chasing them is not feasible, even on small instances. Attendees will learn what features GROM offers to solve this problem. As a first solution, the system will run its greedy chase algorithm to search for solutions to the original deds. This amounts to generating several scenarios made of tgd and egd, that capture specific branches in the deds. This strategy is sound, but not complete. However, it is often surprisingly quick in returning some solution.

In other cases, when the constraints are more intricate, the greedy chase will take considerably more time, due to the fact that many of the generated scenarios fail to generate a solution, and new ones need to be executed. In these cases, a possible alternative is to leverage the syntactic restrictions over the use of negation [9] that guarantee that no deds are generated. In essence, the user needs to inspect the views and change them in such a way to remove perverse negation patterns that will generate deds. GROM supports this process by highlighting problematic views, so that the user may consider alternative formulations.

### 5. REFERENCES


