Supporting Executable Mappings in Model Management

Sergey Melnik Philip A. Bernstein
Microsoft Research
Redmond, WA, U.S.A.
{melnik | philbe}@microsoft.com

Alon Halevy
Univ. of Washington
Seattle, WA, U.S.A.
alon@cs.washington.edu

Erhard Rahm
University of Leipzig
Germany
rahm@informatik.uni-leipzig.de

ABSTRACT

Model management is an approach to simplify the programming of metadata-intensive applications. It offers developers powerful operators, such as Compose, Diff, and Merge, that are applied to models, such as database schemas or interface specifications, and to mappings between models. Prior model management solutions focused on a simple class of mappings that do not have executable semantics. Yet many metadata applications require that mappings be executable, expressed in SQL, XSLT, or other data transformation languages.

In this paper, we develop a semantics for model-management operators that allows applying the operators to executable mappings. Our semantics captures previously-proposed desiderata and is language-independent: the effect of the operators is expressed in terms of what they do to the instances of models and mappings. We describe an implemented prototype in which mappings are represented as dependencies between relational schemas, and discuss algebraic optimization of model-management scripts.

1. INTRODUCTION

Many challenging problems facing information systems engineering involve the manipulation of complex metadata artifacts, or *models*, such as database schemas, ontologies, interface specifications, or workflow definitions, and *mappings* between models, such as SQL views, XSL transformations, or ontology articulations. The applications that solve metadata manipulation problems are complex and hard to build. One reason is due to low-level programming interfaces, which provide access to the individual model elements, such as attribute definitions of database schemas. Programming against such interfaces requires a lot of navigational code. Another reason is that most approaches are language-specific and application-specific, i.e., are developed for SQL, UML, or XML and are not easily portable to other languages or applications.

Model management aims at providing a generic and powerful environment to enable rapid development of metadata-intensive applications [6, 7]. In many of these applications, mappings must be executable on instances of the models, such as databases and messages. Example applications include wrapper generation, message translation, data exchange, and data migration. In this paper, we extend model management to cope with such *executable* mappings, i.e., mappings that compute target instances from source instances, or can be evaluated as constraints on instances.

In the core of the model-management approach is a set of algebraic *operators* that generalize the operations utilized across various metadata applications. The operators are applied to models and mappings as a *whole* rather than to their individual elements, and thus simplify the programming of metadata applications. The operators are designed to be *generic*, i.e., useful for various problems and different kinds of metadata. The major model management operators suggested in the literature include

- Match: create a mapping between two models.
- Compose: combine two successive mappings into one.
- Merge: merge two models into a third model using a mapping between the two models.
- Extract: return a portion of a model that participates in a mapping.
- Diff: return a portion of a model that does not participate in a mapping.

These operators can be used for solving schema evolution, data integration, and other scenarios using short programs, or *scripts* [6, 8]. A script combines the operators so that the outputs produced by one operator can be taken as inputs to another operator. The scripts are executed by a model management system. The first prototype of such a system, called Rondo, was presented in [27].

While the intuition behind the operators is well understood and the operators were shown to be useful for solving practical problems, there exists no implementation of model management that is capable of operating on executable mappings. The reason is primarily due to the lack of rigorous semantics that explains what outputs should be produced if the input mappings are SQL views, XSL transformations, database constraints, etc. And yet, the purpose of many model-management scripts is to generate mappings that drive data migration, message translation, or database wrapping. Such mappings transform instances of models. How does a developer know that a script generates mappings that transform instances as expected? When designing a model-management system, how do we know that our operator implementation is correct? The answers require an understanding of the relationship between the models and mappings returned by each operator and the transformations expressed by those mappings on the states of those models. To explain that relationship, this paper develops a state-based semantics for model management operators. Moreover, we applied the definitions to two concrete languages and built a prototype that supports mappings expressed as constraints using a subset of relational algebra.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-59593-060-4/05/06 $5.00.

Our first contribution is a specification of the semantics of each model management operator on arbitrary models and mappings. The semantics is specified by relating the instances of the operator's input and output models. An *instance* of a model is a state that conforms to the model. For example, an instance of a database schema is a database state, and an instance of an XML schema is an XML document. An instance of a mapping is a tuple of states, one for each of the models involved in the mapping. For example, an instance of a SQL view definition v is a pair (x, y) where x is a database state and y is a state of the view schema computed by v. For the purpose of defining the operators we do not specify the nature of instances any further. We use the terms instance and state interchangeably.

Our second contribution is to study two approaches to specifying and manipulating executable mappings, one based on Rondo [27] and a new approach that operates on mappings represented as logic formulas. Our motivation for using Rondo was to exploit its data structures and algorithms, and to avoid implementing a new system from scratch.[1] Rondo's operators are defined for morphisms, which are sets of uninterpreted correspondences between schema elements. Morphisms are purely syntactic structures that cannot be executed. They are useful in metadata applications that do not require executable mappings, such as model translation (e.g., UML to IDL and even ER to SQL in many cases), dependency tracking, and impact analysis. However, we show that it is possible to assign semantics to a subset of Rondo's language, which we call path-morphisms, and to generate executable mappings from Rondo's outputs under certain assumptions.

Although morphisms are attractive due to their simplicity and natural visual representation, their expressiveness is inherently limited. To overcome this limitation, we developed a new prototype in which mappings are represented as sets of logical dependencies between relational schemas. Our prototype can handle a much broader class of executable mappings than we are able to support by leveraging Rondo. We describe the architecture of the prototype and its representation of models and mappings.

Our implementation efforts indicate that computing the results of a single operator can be very expensive. In fact, some algorithms are exponential in the size of mappings. Even worse, the results of an operator may not be expressible in the supported model and mapping languages, so the script execution fails. As our last contribution, we show that it may be possible to rewrite a model-management script into an equivalent but different script. In certain cases a script that fails can be rewritten as an executable one, and an inefficient one can be optimized.

The individual algorithms, their correctness proofs, and the algebraic rewriting of scripts warrant a rigorous mathematical presentation that is out of the scope of this paper. We lay out the main ideas here and present further details in [28, 26].

The paper is organized as follows. In Section 2, we present a motivating example that illustrates the operators and scripts using a schema/view evolution scenario. Section 3 formally defines models, mappings, and operators. Section 4 defines the semantics of the operators and presents further examples to support the intuition behind our definitions. In Section 5, we focus on the implementation. Section 6 deals with the algebraic rewriting of scripts. Related work is reviewed in Section 7. Section 8 is the conclusion.

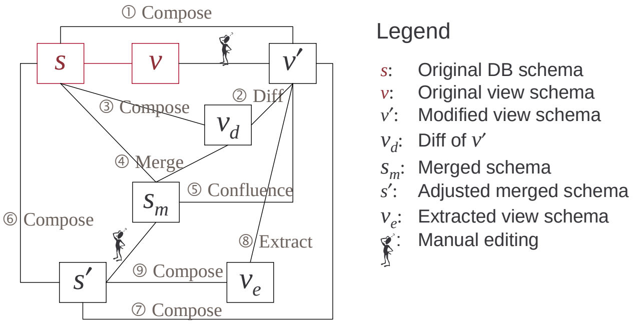


Figure 1: Schematic representation of the solution in motivating example

2. MOTIVATING EXAMPLE

In this section, we present a motivating example that introduces the main model-management operators. We refer to it throughout the paper to explain the features and semantics of the individual operators and to illustrate various technical points.

Consider a typical data management setting in which the data is accessed via user views and is stored in a single integrated database. The database schema is designed to support all deployed user views. Facilitating schema and view evolution in such a setting is challenging, since modifications to schemas and views ripple through the entire data management infrastructure and require painstaking adjustments of schema and view definitions. In the example that we focus on, an update to a view schema triggers a modification of the integrated database schema. This modification prompts the database administrator to revise the schema. As a result, the views defined on the schema, including the updated one, need to be adjusted. We present a way of automating this process using model-management operators and illustrate it step-by-step.

Figure 1 shows a schematic representation of the solution. The rectangles denote schemas, while mappings are shown as edges. Assume that s is the schema of the integrated database that contains two tables: Orders for storing customer orders and Software for keeping the information about the software products and vendors (see Figure 2). Orders has a functional dependency Customer → Name (not shown in the figure). Let s_v be a view with view schema v that is used to query the database. The view selects all customers that bought ACME's products and is defined as

$$v.\mathrm{M} = \pi_{\mathrm{C,N}}(\sigma_{\mathrm{V=``ACME"}}(s.\mathrm{O} \bowtie s.\mathrm{S}))$$

Table and column names are abbreviated using their initial letters. The above and the following mapping expressions are summarized for convenience in Figure 3.

Suppose that the view schema v has been edited to include the columns DOB ("date of birth") and IsCorporate for supporting a new business policy for corporate customers. The relationship between v and the new view schema v' is specified by the mapping v_v' produced as part of output of the schema editor:

$$v.\mathrm{M} = \pi_{\mathrm{C,N}}(v'.\mathrm{M})$$

The database schema s needs to be augmented to store the new information queryable via the new view schema. To do so, we need to add to s the new schema elements that appear in v' and update the view definition. This task can be accomplished using the following model-management script, which we explain line by line below:

1. $s\_v' = s\_v \circ v\_v'$
2. $\langle v_d, v'\_v_d \rangle = \mathsf{Diff}(v', \mathsf{Invert}(s\_v'))$
3. $s\_v_d = s\_v' \circ v'\_v_d$
4. $\langle s_m, s_m\_s, s_m\_v_d \rangle = \mathsf{Merge}(s, v_d, s\_v_d)$
5. $s_m\_v' = (s_m\_s \circ s\_v') \oplus (s_m\_v_d \circ \mathsf{Invert}(v'\_v_d))$

[1] The source code and sample scripts of Rondo are available at http://www-db.stanford.edu/∼modman/rondo/

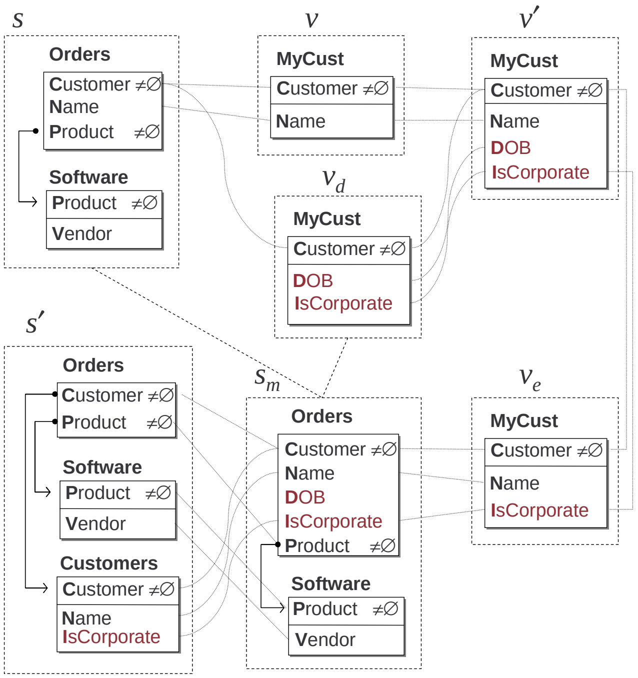


Figure 2: Schemas in motivating example

1. In Line 1, we determine the relationship s_v' between s and v' by computing the composition of mappings s_v and v_v':

$$\pi_{\mathrm{C,N}}(\sigma_{\mathrm{V=``ACME"}}(s.\mathrm{O} \bowtie s.\mathrm{S})) = \pi_{\mathrm{C,N}}(v'.\mathrm{M})$$

2. In Line 2, the operator Diff is applied to determine the new information that v' adds to v, i.e., the portion of v' that does not participate in the mapping s_v'. The operator Diff expects a mapping between v' and s. Therefore the operator Invert is applied prior to Diff to swap the left and right sides of the input mapping. Diff produces the "difference" schema v_d and a mapping v'_v_d that relates it to v' (see Figure 3).

3. Line 3 shows another composition that gives us the mapping s_v_d between s and v_d, which is needed to merge s and v_d in the next step.

4. The operator Merge takes as input the original database schema s, the new portion of the view schema v_d, and the mapping s_v_d that identifies "overlapping" information in s and v_d, which needs to be represented only once in the merged schema s_m. As a result, columns DOB and IsCorporate are added to the table Orders in s_m. The output mappings s_m_s and $s_m\_v_d$ describe how the merged schema relates to the input schemas (mappings not shown in Figure 2).

5. In Line 5, we reconstruct the view definition between the merged schema s_m and the new view schema v'. As can be seen in Figure 1, there exist two "paths" connecting s_m and v': one via s and another one via v_d. The first "path" relates the unchanged schema elements in s_m and v', and is given by the mapping composition $s_m\_s \circ s\_v'$:

$$\pi_{\mathrm{C,N}}(\sigma_{\mathrm{V=``ACME"}}(s_m.\mathrm{O} \bowtie s_m.\mathrm{S})) = \pi_{\mathrm{C,N}}(v'.\mathrm{M})$$

The second "path" relates the new elements and is given by the composition $s_m\_v_d \circ \mathsf{Invert}(v'\_v_d)$:

$$\pi_{\mathrm{C,D,I}}(\sigma_{\mathrm{V=``ACME"}}(s_m.\mathrm{O} \bowtie s_m.\mathrm{S})) = \pi_{\mathrm{C,D,I}}(v'.\mathrm{M})$$

Produced by	Mapping expression
🧍	s_v: $\pi_{\mathrm{C,N}}(\sigma_{\mathrm{V="ACME"}}(s.\mathrm{O}\bowtie s.\mathrm{S})) = v.\mathrm{M}$
	v_v': $v.\mathrm{M} = \pi_{\mathrm{C,N}}(v'.\mathrm{M})$
① Compose	s_v': $\pi_{\mathrm{C,N}}(\sigma_{\mathrm{V="ACME"}}(s.\mathrm{O}\bowtie s.\mathrm{S})) = \pi_{\mathrm{C,N}}(v'.\mathrm{M})$
② Diff	v'_v_d: $\pi_{\mathrm{C,D,I}}(v'.\mathrm{M}) = v_d.\mathrm{M}$
③ Compose	s_v_d: $\pi_{\mathrm{C}}(\sigma_{\mathrm{V="ACME"}}(s.\mathrm{O}\bowtie s.\mathrm{S})) = \pi_{\mathrm{C}}(v_d.\mathrm{M})$
④ Merge	s_m_s: $\pi_{\mathrm{C,N,P}}(s_m.\mathrm{O}) = s.\mathrm{O}$ $s_m.\mathrm{S} = s.\mathrm{S}$
④ Merge	$s_m\_v_d$: $\pi_{\mathrm{C,D,I}}(\sigma_{\mathrm{V="ACME"}}(s_m.\mathrm{O}\bowtie s_m.\mathrm{S})) = v_d.\mathrm{M}$
🧍	s'_s_m: $s'.\mathrm{O}=\pi_{\mathrm{C,P}}\,(s_m.\mathrm{O})$ $s'.\mathrm{S} = s_m.\mathrm{S}$ $s'.\mathrm{C} = \pi_{\mathrm{C,N,I}}(s_m.\mathrm{O})$
⑤ Confluence	s_m_v': $\pi_{\mathrm{C,N,D,I}}(\sigma_{\mathrm{V="ACME"}}(s_m.\mathrm{O}\bowtie s_m.\mathrm{S})) = v'.\mathrm{M}$
⑥ Compose	s'_s: $s'.\mathrm{O}=\pi_{\mathrm{C,P}}\,(s.\mathrm{O})$ $s'.\mathrm{S} = s.\mathrm{S}$ $\pi_{\mathrm{C,N}}(s'.\mathrm{C}) = \pi_{\mathrm{C,N}}(s.\mathrm{O})$
⑦ Compose	s'_v': $\pi_{\mathrm{C,N,I}}(\sigma_{\mathrm{V="ACME"}}(s'.\mathrm{O}\bowtie s'.\mathrm{S}\bowtie s'.\mathrm{C})) = \pi_{\mathrm{C,N,I}}(v'.\mathrm{M})$
⑧ Extract	v'_v_e: $\pi_{\mathrm{C,N,I}}(v'.\mathrm{M}) = v_e.\mathrm{M}$
⑨ Compose	s'_v_e: $\pi_{\mathrm{C,N,I}}(\sigma_{\mathrm{V="ACME"}}(s'.\mathrm{O}\bowtie s'.\mathrm{S}\bowtie s'.\mathrm{C})) = v_e.\mathrm{M}$

Figure 3: Mappings in motivating example

The operator Confluence (denoted by $\oplus$) is used to create a single mapping s_m_v' from the two partial mappings above. Exploiting functional dependency C $\rightarrow$ {N,D,I}, we get:

$$\pi_{\mathrm{C,N,D,I}}(\sigma_{\mathrm{V=``ACME"}}(s_m.\mathrm{O} \bowtie s_m.\mathrm{S})) = v'.\mathrm{M}$$

After running the above script, the database administrator inspects the merged schema s_m and realizes that table Orders contains substantial redundancy, because customer data is repeated in each order, so she normalizes the schema: customers are moved to a separate table Customers. The administrator also discovers that storing personal information such as date of birth violates the company's privacy policy; she deletes the new column DOB. The relationship between the original merged schema s_m and the revised schema s' is specified by the mapping s'_s_m.

To migrate the data from the currently deployed schema s to the revised schema s', the administrator computes the mapping s'_s as

6. $s'\_s = s'\_s_m \circ s_m\_s$

The mapping s'_s is used to automatically derive a set of data migration operations that include creating a new table Customers and deleting customer Name from Orders.

All views defined on s need to be updated to operate on the revised merged schema s'. We present the script for updating the view v' that triggered the modification of s. To update any other view v_i on s, simply replace s_m by s and v' by v_i in the script:

7. $s'\_v' = s'\_s_m \circ s_m\_v'$
8. $\langle v_e, v'\_v_e \rangle = \mathsf{Extract}(v', \mathsf{Invert}(s'\_v'))$
9. $s'\_v_e = s'\_v' \circ v'\_v_e$

7. In Line 7, we obtain the mapping between the revised schema s' and the view v' by composing the mapping s'_s_m that captures the schema revisions with the existing view definition s_m_v'. The mapping s'_v' is computed by exploiting the PK/FK constraints in schemas s', s_m, and v'.

8. In Line 8, we use the operator Extract to eliminate the "dangling" schema element DOB from the view schema v' that has

been deleted from s_m. Extract returns the portion v_e of the view schema v' that participates in the mapping s'_v' and the mapping v'_v_e that ties it back to v'.

9. As a last step, we reconstruct the view definition by composing s'_v' obtained in Line 7 with the output mapping of Extract. The composition yields the updated view definition

$$\pi_{\mathrm{C,N,I}}(\sigma_{\mathrm{V=\text{“ACME”}}}(s'.\mathrm{O} \bowtie s'.\mathrm{S} \bowtie s'.\mathrm{C})) = v_e.\mathrm{M}$$

The above script applied model-management operators to executable mappings, thereby automating manipulation of view definitions and dependencies between schemas. Similar kinds of scripts can be used to solve other problems, such as reverse engineering [6] or 3-way merge [25]. In the subsequent sections, we define the semantics of the operators. The prototype that we built operates on relational algebra expressions such as the ones shown in Figure 1, but our definitions of operator semantics are applicable to any data transformation language.

3. PREREQUISITES

We present the basic concepts of model management, including models, mappings, operators, and scripts, and explain the notation used in the paper. In the examples, we use relational schemas and assume set semantics for the relations and queries. In our discussion, we use the terms query and view synonymously. More precisely, a view is a named query, whose result schema, called view schema, is specified explicitly.

Models. A *model* is a set of instances. Sometimes, a model can be denoted by an expression in a concrete language, such as SQL DDL, XML Schema, BPEL4WS [5], or CORBA IDL [35]. For example, a relational schema denotes a set of database states; a workflow definition denotes a set of workflow instances; a programming interface denotes a set of implementations that conform to the interface. To refer to models, we use variables s, v, m, m_1, m_2, etc. When x is an instance of model m, we write $x \in m$. When we use an expression to denote a model, we put it in French quotation marks, such as ≪E≫ for expression E.

EXAMPLE 1 Each instance of the view schema v_e = ≪MyCust(<u>Customer</u>, Name, IsCorporate)≫ in Figure 2 is an entire materialized view containing a populated relation MyCust. □

Mappings. A *mapping* is a relation on instances. In this paper, we focus on binary mappings, i.e., those that hold between two models. Sometimes, a mapping can be denoted using an expression in a concrete language such as relational algebra, SQL DML, XSLT, GLAV, etc. We put such expressions in French quotation marks.[2] To refer to mappings, we use variables map_1, map_2, $m_1\_m_2$, $m_2\_m_3$, etc.

EXAMPLE 2 The mapping

$$s\_v' = \ll\pi_{\mathrm{C,N}}(\sigma_{\mathrm{V=\text{“ACME”}}}(s.\mathrm{O} \bowtie s.\mathrm{S})) = \pi_{\mathrm{C,N}}(v'.\mathrm{M})\gg$$

is a binary relation on instances of s and v' of Figure 2. That is, for all databases $x \in s$, $y \in v'$: $(x, y) \in s\_v'$ if and only if the mapping constraint is satisfied for x and y. □

A mapping can be thought of as a constraint that holds between two models [9, 22, 23]. If $m_1\_m_2 = m_1 \times m_2$, the constraint is empty; if $m_1\_m_2 = \emptyset$, it is contradictory. In general, $m_1\_m_2$ is an arbitrary binary relation on instances, which may be total, partial, functional, surjective, etc. A *query* is a partial functional mapping [1]. A query (and hence a mapping in general) may not be expressible in a specific query language.

If $(x, y) \in m_1\_m_2$ we say that instances x and y are *consistent* under $m_1\_m_2$, i.e., can co-exist simultaneously in the application that deploys the mapping $m_1\_m_2$. If each instance of m_1 is consistent under $m_1\_m_2$ with at least one instance in m_2 and vice versa, we call models m_1 and m_2 consistent under $m_1\_m_2$ (or conflict-free as in [9]). That is, m_1 and m_2 are consistent under $m_1\_m_2$ iff $m_1\_m_2$ is a total surjective mapping between m_1 and m_2.

Operators. A model-management *operator* takes models and mappings as input and produces models and mappings as output. Formally, a model-management operator is an n-ary predicate on models and mappings. The attributes of the predicate are partitioned into input attributes and output attributes. (We define operators as predicates because they may produce different outputs that are all consistent with the definition of the operators.)

To compute the outputs of an operator effectively for all inputs, the following property is essential:

DEFINITION 1 (OPERATOR CLOSURE) *Let $\mathcal{L}$ be a language for specifying models and mappings, and let θ be a model-management operator. $\mathcal{L}$ is* closed under θ *if given any inputs to θ in $\mathcal{L}$ the outputs can also be expressed in $\mathcal{L}$.* □

Scripts. A model-management *script* is a conjunctive formula built from model-management operators. The variables and constants in a script refer to models and mappings. Computing the script means finding a valid substitution, which is one that replaces all variables by constants (i.e., concrete model and mapping definitions) and makes the script a true formula. For example, substituting the variables in the script presented in Section 2 by the schemas and mappings shown in Figures 2-3 yields a true formula.

4. OPERATORS

In this section we suggest a state-based semantics for the six major operators proposed in the literature [6, 7, 8, 27]: Match, Compose ($\circ$), Merge, Extract, Diff, and Confluence ($\oplus$), as well as five auxiliary operators: cross-product, Id, Invert, Domain, and Range.

Of all the operators, Match plays a special role. Given two models m_1 and m_2, the operator returns a mapping $m_1\_m_2$ that holds between the models, denoted as $m_1\_m_2 = \mathsf{Match}(m_1, m_2)$. The operator Match inherently does not have formal semantics. It gives us what we know about the relationship between models in a particular application context. Sometimes this relationship can be discovered semi-automatically [31] but ultimately Match depends on human feedback (and hence may be partial or even inaccurate).

For the auxiliary operators and the composition operator we adopt the standard algebraic definitions:

- $m_1 \times m_2 =_{\mathrm{df}} \{(x, y) \mid x \in m_1 \text{ and } y \in m_2\}$
- $\mathsf{Invert}(map) =_{\mathrm{df}} \{(y, x) \mid (x, y) \in map\}$
- $\mathsf{Domain}(map) =_{\mathrm{df}} \{x \mid \exists y : (x, y) \in map\}$
- $\mathsf{Range}(map) =_{\mathrm{df}} \mathsf{Domain}(\mathsf{Invert}(map))$
- $\mathsf{Id}(m) =_{\mathrm{df}} \{(x, x) \mid x \in m\}$

[2] To be well-defined, mapping expressions need to include the definitions of the models they reference (as done in [16, 28]); we omit them here for brevity.

- $map_{12} \circ map_{23} =_{\mathrm{df}} \{(x,z) \mid \exists y : (x,y) \in map_{12} \wedge (y,z) \in map_{23}\}$

Thus, many well-known properties hold, such as

1. $\mathsf{Domain}(\mathsf{Id}(m)) = m$
2. $\mathsf{Invert}(\mathsf{Invert}(map)) = map$
3. $map_1 \circ (map_2 \circ map_3) = (map_1 \circ map_2) \circ map_3$
4. $map_1 \circ map_2 = \mathsf{Invert}(\mathsf{Invert}(map_2) \circ \mathsf{Invert}(map_1))$
5. Mapping map is a surjective function onto m if and only if $\mathsf{Invert}(map) \circ map = \mathsf{Id}(m)$

The definitions of Merge, Extract, and Diff given below capture the necessary conditions on the operators. However, they allow the operators to produce non-unique output models and mappings. Traditionally, in problem settings that have this property, various *optimality criteria* have been used to drive the computation of the outputs, including query evaluation cost [2], size of schema instances [21, 38], or syntactic properties, such as size of expressions used for schemas and queries [9, 36]. In Section 6, we discuss an additional minimality requirement that facilitates algebraic rewriting and optimization of scripts.

4.1 Merge operator

In the motivating example of Section 2, we use the operator Merge to incorporate the new schema elements of v_d into an existing schema s. One obvious requirement for this task is to preserve all information in the input schemas. And yet, since column Customer appears in both schemas, our intuition suggests that it is sufficient to represent it only once in the merged schema.

These desiderata strongly resemble those of view integration, if we consider s and v_d to be virtual views for which an integrated schema needs to be produced. We motivate our definition of Merge by the view integration problem, as explained in the following scenario:

EXAMPLE 3 Consider a company with two departments, each of which manages its own database. Let m_1 and m_2 be the respective database schemas (see Figure 4). Suppose m_1 and m_2 are not disjoint; for instance, both describe employee data. The mapping $m_1\_m_2$ describes the mutually consistent states of m_1 and m_2.

To simplify the management of data across the departmental databases, the company decides to move all data to a centralized database, which the departments access using view schemas m_1 and m_2. Thus, the goal is to create a schema m for the centralized database and views m_m_1 and m_m_2, such that m captures all the information needed by the departments. If the autonomy of the departments needs to be restored later on, it should be possible to reconstruct m_1, m_2, and $m_1\_m_2$ from m, m_m_1, and m_m_2, i.e., the transition to the centralized database must not lose information. □

The following is the formal definition of Merge, which captures the properties of the above scenario and our desiderata for the motivating example.

DEFINITION 2 (MERGE) *Let $m_1\_m_2$ be a mapping between m_1 and m_2. $\langle m, m\_m_1, m\_m_2 \rangle = \mathsf{Merge}(m_1, m_2, m_1\_m_2)$ holds only if*

i. m_m_1 and m_m_2 are (possibly partial) surjective functions onto m_1 and m_2, respectively

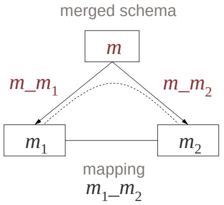


Figure 4: Illustration of Merge (Example 3)

ii. $m_1\_m_2 = \mathsf{Invert}(m\_m_1) \circ m\_m_2$

iii. $m = \mathsf{Domain}(m\_m_1) \cup \mathsf{Domain}(m\_m_2)$ □

Condition (i) states that m_m_1 and m_m_2 are views on m. Due to surjectivity, $m_1 = \mathsf{Range}(m\_m_1)$ and $m_2 = \mathsf{Range}(m\_m_2)$, so m contains all the information of m_1 and m_2. Condition (ii) guarantees that the input mapping $m_1\_m_2$ can be reconstructed from the views. That is, we can obtain the mutually consistent states of m_1 and m_2 by the composition $\mathsf{Invert}(m\_m_1) \circ m\_m_2$. Condition (iii) ensures that the information in m is used either in view m_m_1 or view m_m_2.

EXAMPLE 4 In the motivating example, Merge is used in Line 4 of the script:

4. $\langle s_m, s_m\_s, s_m\_v_d \rangle = \mathsf{Merge}(s, v_d, s\_v_d)$

The input mapping is given as

$$s\_v_d = \ll \pi_{\mathrm{C}}(\sigma_{\mathrm{V=“ACME”}}(s.\mathrm{O} \bowtie s.\mathrm{S})) = \pi_{\mathrm{C}}(v_d.\mathrm{M}) \gg$$

The operator produces the mappings

$$s_m\_s = \ll \pi_{\mathrm{C,N,P}}(s_m.\mathrm{O}) = s.\mathrm{O};\ s_m.\mathrm{S} = s.\mathrm{S} \gg$$
$$s_m\_v_d = \ll \pi_{\mathrm{C,D,I}}(\sigma_{\mathrm{V=“ACME”}}(s_m.\mathrm{O} \bowtie s_m.\mathrm{S})) = v_d.\mathrm{M} \gg$$

Clearly, s_m_s and $s_m\_v_d$ are total surjective views: we can reconstruct the state of s and v_d uniquely given a snapshot of s_m. Moreover, each state of s and v_d is covered by some state of s_m, so condition (i) holds. The views are total, hence condition (iii) is satisfied. Consequently, we can represent all information of s and v_d using s_m.

The output mappings s_m_s and $s_m\_v_d$ allow us to reconstruct the constraints between s and v_d by composition as $s\_v_d = \mathsf{Invert}(s_m\_s) \circ s_m\_v_d$. The mapping $s_m\_v_d$ has a join condition, which legitimately requires s_m to have values for DOB and IsCorporate only for customers who purchased ACME's products. In contrast, a more straightforward $\pi_{\mathrm{C,D,I}}(s_m.\mathrm{O}) = v_d.\mathrm{M}$ would incorrectly place this requirement on all customers. Condition (ii) prohibits such a straightforward mapping. □

4.2 Extract operator

In Line 7 of the script in the motivating example, we obtain the mapping s'_v' that links the modified view schema v' and the database schema s' adjusted by the administrator:

$$\pi_{\mathrm{C,N,I}}(\sigma_{\mathrm{V=“ACME”}}(s'.\mathrm{O} \bowtie s'.\mathrm{S} \bowtie s'.\mathrm{C})) = \pi_{\mathrm{C,N,I}}(v'.\mathrm{M})$$

Observe that mapping s'_v' is not functional: it does not tell us how to obtain DOB values in the view relation $v'.\mathrm{M}$ from the database. To retain just the portion of the view that we know how to populate from the database, we use the operator Extract. The operator embodies the primary correctness requirement of view selection in data warehousing, namely, that of preserving the query-answering capability of the selected view. To illustrate, consider the following scenario:

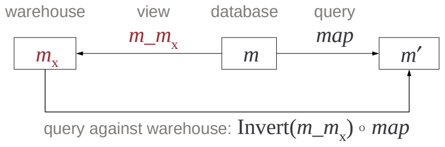


Figure 5: Illustration of Extract (Example 5)

EXAMPLE 5 Let m be a database schema and map be a query over m. Our goal is to select a set of view relations to materialize in a data warehouse. The query map against m must be answerable using the warehouse schema m_x (see Figure 5). The view definition m_m_x tells us how to populate the warehouse from the database. We can reformulate our original query map to run against m_x by composing the reverse transformation $\mathsf{Invert}(m\_m_x)$ and map. □

The following definition describes formally the properties of m_x and m_m_x in the above example.

DEFINITION 3 (EXTRACT) *Let map be a mapping from m. $\langle m_x, m\_m_x\rangle = \mathsf{Extract}(m, map)$ holds only if*

i. $m\_m_x \circ \mathsf{Invert}(m\_m_x) \circ map = map$

ii. $m_x = \mathsf{Range}(m\_m_x)$ □

In the definition, m_m_x is the database transformation from m to the new schema m_x, while $\mathsf{Invert}(m\_m_x) \circ map$ is the updated query over m_x. Hence, condition (i) requires the updated query over m_x to produce the same results as the original query map over m. Condition (ii) makes the output model m_x to be the range (i.e., view schema) of m_m_x.

Extract returns a portion m_x of m that is "observable" through mapping map. If map is a query, Definition 3 can be satisfied by choosing $m_x = m'$, $m\_m_x = map$. In fact, we can always create a data warehouse by directly using the query workload map. Furthermore, for any map we can populate the warehouse by copying the entire database, i.e., a trivial implementation of Extract is $m_x = m$ and $m\_m_x = \mathsf{Id}(m)$. In general, however, it may be possible to construct more compact solutions that still satisfy conditions (i)-(ii), as illustrated below.

EXAMPLE 6 Consider again the motivating example of Section 2. Extract is used in Line 8:

8. $\langle v_e, v'\_v_e\rangle = \mathsf{Extract}(v', \mathsf{Invert}(s'\_v'))$

and yields the output

$v_e = \ll \mathsf{MyCust}(\underline{\mathsf{Customer}}, \mathsf{Name}, \mathsf{IsCorporate})\gg$
$v'\_v_e = \ll \pi_{\mathrm{C,N,I}}(v'.\mathrm{M}) = v_e.\mathrm{M}\gg$

The view schema v_e contains precisely those columns of v' that are referenced in the input mapping s'_v'. Their presence in v_e is required to satisfy condition (i):

$v'\_v_e \circ \mathsf{Invert}(v'\_v_e) \circ \mathsf{Invert}(s'\_v') = \mathsf{Invert}(s'\_v')$

The condition is not satisfied trivially, since the composition $v'\_v_e \circ \mathsf{Invert}(v'\_v_e)$ does not yield an identity mapping. Instead, the composition produces a mapping that groups the database states of v' by the columns Customer, Name, and IsCorporate. These are exactly the columns referenced in the mapping s'_v', so the above equality holds. □

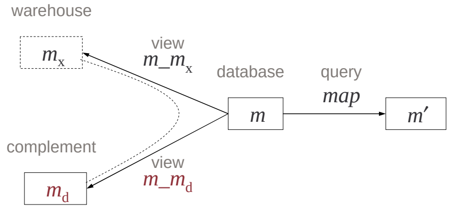


Figure 6: Illustration of Diff (Example 7)

4.3 Diff operator

In Section 2, the operator Diff is used to compute the "difference" between the original view schema v and its modified version v'. To motivate the formal definition, we draw upon the problem of computing view complements in data warehousing. We continue with the scenario of Example 5.

EXAMPLE 7 Updates of the materialized view relations in the warehouse m_x generate a heavy load on the database m due to maintenance queries. To offload the database, we set up a complementary database with schema m_d, which stores the portion of the data of m that is not in m_x. The view m_m_d populates m_d with data from m. Together, m_m_d and m_m_x describe how the data in the warehouse relates to the data in the complementary database, namely that $m_x\_m_d = \mathsf{Invert}(m\_m_x) \circ m\_m_d$. The original database can be reconstructed by merging m_x and m_d based on $m_x\_m_d$. □

The following definition specifies formally the properties of m_d and m_m_d in the above example:

DEFINITION 4 (DIFF) *Let map be a mapping from m. $\langle m_d, m\_m_d\rangle = \mathsf{Diff}(m, map)$ holds only if for any m_x, m_m_x satisfying*

$\langle m_x, m\_m_x\rangle = \mathsf{Extract}(m, map)$

the following condition holds:

$\langle m, m\_m_x, m\_m_d\rangle =$
$\mathsf{Merge}(m_x, m_d, \mathsf{Invert}(m\_m_x) \circ m\_m_d)$ □

Notice that Diff has a trivial solution $m_d = m$, $m\_m_d = \mathsf{Id}(m)$, which we obtain by copying the whole database. However, often it is possible to construct a result that has less redundancy:

EXAMPLE 8 In the motivating example, Diff is used in:

2. $\langle v_d, v'\_v_d\rangle = \mathsf{Diff}(v', \mathsf{Invert}(s\_v'))$

with the following inputs and outputs:

$v' = \ll \mathsf{MyCust}(\underline{\mathsf{Customer}}, \mathsf{Name}, \mathsf{DOB}, \mathsf{IsCorporate})\gg$
$s\_v' = \ll \pi_{\mathrm{C,N}}(\sigma_{\mathrm{V=\text{“ACME”}}}(s.\mathrm{O} \bowtie s.\mathrm{S})) = \pi_{\mathrm{C,N}}(v'.\mathrm{M})\gg$
$v_d = \ll \mathsf{MyCust}(\underline{\mathsf{Customer}}, \mathsf{DOB}, \mathsf{IsCorporate})\gg$
$v'\_v_d = \ll \pi_{\mathrm{C,D,I}}(v'.\mathrm{M}) = v_d.\mathrm{M}\gg$

The difference schema v_d contains the two columns DOB and IsCorporate that do not appear in the mapping s_v'. In addition, it contains the primary key Customer. The key column is needed to link the tuples of v_d to those of v'. The view v'_v_d complements the mapping $\mathsf{Invert}(s\_v')$. That is, v_d and v'_v_d satisfy Definition 4 for any extracted v_x and v'_v_x. In particular, for

$v_x = \ll \mathsf{MyCust}(\underline{\mathsf{Customer}}, \mathsf{Name})\gg$
$v'\_v_x = \ll \pi_{\mathrm{C,N}}(v'.\mathrm{M}) = v_x.\mathrm{M}\gg$
$v_x\_v_d = \ll \pi_{\mathrm{C}}(v_x.\mathrm{M}) = \pi_{\mathrm{C}}(v_d.\mathrm{M})\gg$

it is easy to check that

$$
\begin{aligned}
&v_x\_v_d = \mathsf{Invert}(v'\_v_x) \circ v'\_v_d\\
&\langle v_x, v'\_v_x\rangle = \mathsf{Extract}(v', \mathsf{Invert}(s\_v'))\\
&\langle v', v'\_v_x, v'\_v_d\rangle = \mathsf{Merge}(v_x, v_d, v_x\_v_d)
\end{aligned}
$$

Outputs v_d and v'_v_d yield a correct "difference"; in contrast, dropping the key column from v_d produces a lossy complement. □

4.4 Confluence operator

Confluence is a new operator that we developed by analyzing the properties of several model-management scenarios, such as change propagation [6, 27]. It "unifies" two partial or possibly inconsistent mappings map_1 and map_2 between models m_1 and m_2. Mappings map_1 and map_2 may have been designed independently by two engineers, or obtained as results of other model-management operators. In effect, Confluence merges two mappings, as opposed to merging models. It is defined as follows:

DEFINITION 5 (CONFLUENCE, $\oplus$)

$$
\begin{aligned}
&map_1 \oplus map_2 =_{\mathrm{df}} (map_1 \cap map_2)\\
&\quad \cup \{(x, y) \in map_1 \mid x \notin \mathit{Domain}(map_2) \wedge y \notin \mathit{Range}(map_2)\}\\
&\quad \cup \{(x, y) \in map_2 \mid x \notin \mathit{Domain}(map_1) \wedge y \notin \mathit{Range}(map_1)\}
\end{aligned}
$$

In the motivating example of Section 2, the mappings that are inputs to Confluence agree on the domain and range (see explanation for Line 5). In this case, the result of confluence is a conjunction of the constraints in the input mappings, i.e., $map_1 \oplus map_2 = map_1 \cap map_2$. We touch upon the general case in Section 5.2.

5. IMPLEMENTATION

To implement the operators for executable mappings, we are pursuing two complementary approaches. The first leverages the implementation of [27], in which mappings are represented as convenient data structures, called morphisms. Morphisms are attractive because of their simplicity, intuitive visualization, and ease of programming. The algorithms that operate on morphisms are highly efficient. On the downside, morphisms have limited expressive power. Thus, as a more general approach, we developed a new prototype in which mappings are represented as calculus expressions. In the following subsections we elaborate on the two techniques.

5.1 Implementing mappings for Rondo

One way to gain the benefits of executable mappings is to reuse an existing implementation of model management provided by Rondo [27]. It precisely specifies the metadata artifacts produced as output by its operators, but it does not specify a semantics for its mapping language, called morphisms. We developed a state-based semantics for a subset of that language, called path-morphisms, which enables us to generate executable mappings from the outputs of Rondo scripts.[3] To illustrate our approach, we present a restrictive interpretation of morphisms, and mention some extensions.

We start with some preliminary definitions: A *morphism* is a set of pairs of *elements* of two schemas, such as XML attributes or relational tables. For example, the morphism denoted by the arcs between schemas v and v' in Figure 2 is given by

$$\ll\langle v.\mathrm{M.C}, v'.\mathrm{M.C}\rangle, \langle v.\mathrm{M.N}, v'.\mathrm{M.N}\rangle\gg$$

We define path-morphisms for a subset of relational schemas that we call tree schemas: Let graph G_s of a relational schema s be a directed graph whose nodes are relations and vertices are foreign-key to primary-key (FK-PK) relationships. Schema s is a *tree schema* if (i) G_s is a forest of trees, (ii) each relation in s has a primary key, and (iii) each FK-PK join is lossless. Essentially, each tree in a tree schema is a nested relation, or a snowflake schema as used in data warehousing. Lossless FK-PK joins ensure that all tuples in the PK-table are referenced from the FK-table.

The *root key* K_T of tree T is the primary key of the root relation of T. Analogously, the root key K_e of a schema element e is the root key K_T of the tree T that contains e. Trees T_1, T_2 are *connected* by morphism map if map contains an arc between an element of T_1 and an element of T_2. We define path-morphisms as follows:

DEFINITION 6 (PATH-MORPHISM) *Let map be a morphism connecting tree schemas m_1 and m_2. If map connects each tree of one schema to at most one tree of the other schema and map connects the root keys of every pair of connected trees, then map is a* path-morphism.

The algorithm *MorphEx* shown below generates a relational algebra expression Σ for a path-morphism map. The algorithm exploits the fact that there is at most one join path between every two relations in a tree schema.

```
Algorithm MorphEx($map$)
  $\Sigma := \{\}$;
  for each pair of attributes $\langle e_1, e_2\rangle \in map$ do
    $K_1$ := root key for $e_1$;
    $K_2$ := root key for $e_2$;
    $path_1$ := RelationOf($e_1$) $\bowtie \cdots \bowtie$ RelationOf($K_1$);
    $path_2$ := RelationOf($e_2$) $\bowtie \cdots \bowtie$ RelationOf($K_2$);
    $\Sigma = \Sigma \cup \{\pi_{K_1,e_1}(path_1) = \pi_{K_2,e_2}(path_2)\}$;
  end for
return $\Sigma$
```

EXAMPLE 9 Consider again relational schemas v and v' in Figure 2. Let map be the morphism between the schemas given by the two arcs that connect the schemas. It is easy to verify that both schemas are tree schemas and map is a path-morphism such that Σ is the conjunction of the following individual constraints:

1. $\pi_{\mathrm{C}}(v.\mathrm{M}) = \pi_{\mathrm{C}}(v'.\mathrm{M})$
2. $\pi_{\mathrm{C,N}}(v.\mathrm{M}) = \pi_{\mathrm{C,N}}(v'.\mathrm{M})$

Constraint (1) is entailed by (2) and is redundant. □

Let $\mathcal{L}_P$ be the language whose schemas are tree schemas and mappings are path-morphisms. Although $\mathcal{L}_P$ has limited expressiveness, it can represent schemas and mappings in many practical change propagation and schema evolution scenarios. Moreover, the definition of tree schemas and path-morphisms can be easily extended to XML tree schemas in which XML types correspond to relational tables and type references play the role of PK-FK dependencies.

Figure 7 illustrates how Rondo can be applied to obtain executable mappings. The following proposition states the conditions under which it is possible. Since Match has no formal semantics, we assume that Match is used to obtain all morphisms required as input prior to executing the script:

PROPOSITION 1 If the schemas and morphisms that are inputs to a script are in $\mathcal{L}_P$ and are closed under Compose, Confluence, and Invert, then the mapping expressions generated by the *MorphEx* algorithm from the output models and mappings of Rondo satisfy the state-based semantics of Section 4. □

[3] In addition to schemas and morphisms, Rondo uses a third structure called selectors. We treat selectors as identity morphisms on subsets of schema elements.

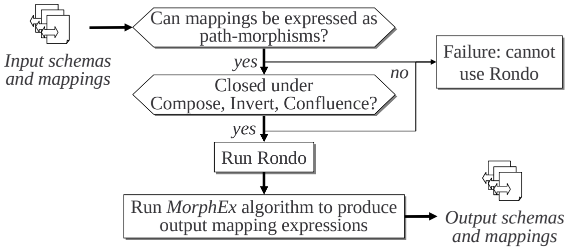

Input schemas and mappings
Can mappings be expressed as path-morphisms?
yes
no
Failure: cannot use Rondo
Closed under Compose, Invert, Confluence?
yes
Run Rondo
Run *MorphEx* algorithm to produce output mapping expressions
Output schemas and mappings

Figure 7: Using Rondo to compute executable mappings

The closure criterion (see Definition 1) requires that the composition and confluence of the input path-morphisms (and their inverses) be expressible as path-morphisms. It can be effectively tested by enumerating all compositions and confluences of pairs of non-inverted and inverted input mappings and checking that each of them yields a path-morphism. If the closure property holds, the expressions produced by the *MorphEx* algorithm are correct and can be deployed in metadata applications to do constraint checking or data migration (if the output mappings are functional).

For 1:1 path-morphisms whose only connected keys are root keys we can make an even stronger claim: whenever the input morphisms fall into this class, we can always compute the correct mapping expressions from the results of scripts. It works because in this case the conjunction of the constraints in the input mappings does not imply any within-schema constraints, which would not be expressible as a morphism.

To appreciate the positive results that we presented above for path-morphisms, consider a simple extension in which we do not require path-morphisms to connect root keys.

EXAMPLE 10 Let morphisms map_{12}, map_{23} and their relational algebra expressions be given as

$$m_1 = \ll\mathrm{R(A)}\gg,\quad m_2 = \ll\mathrm{S(\underline{A},B)}\gg,\quad m_3 = \ll\mathrm{T(B)}\gg$$
$$map_{12} = \ll\langle\mathrm{R.A, S.A}\rangle\gg = \ll\mathrm{R} = \pi_{\mathrm{A}}(\mathrm{S})\gg$$
$$map_{23} = \ll\langle\mathrm{S.B, T.B}\rangle\gg = \ll\mathrm{T} = \pi_{\mathrm{B}}(\mathrm{S})\gg$$

Notice that map_{23} is not a path-morphism. The mapping constraints and the primary key constraint on S imply that S is a function from R onto T. That is, composition $map_{12} \circ map_{23}$ must specify that R contains at least as many different values as T. This constraint is not expressible by a path-morphism, nor by any finite first-order formula. □

We found that extending the interpretation of morphisms to cover more expressive classes of mapping expressions is quite challenging. It is also rather important, since many scenarios require richer expressions than path-morphisms. For example, in the motivating scenario of Section 2, Customer is not a key of s, s_m, or s', so s_v, s_s_m, and s'_v_e are not path-morphisms.

5.2 Moda: a new prototype for model management

One approach to supporting richer expressions is to represent mappings directly as logic formulas, instead of encoding them in morphisms. We therefore implemented a new model management system, Moda. In Moda, models are relational schemas, and mappings are logic formulas that express executable data transformations (as opposed to the graph schemas and morphisms of Rondo). The core of Moda is an implementation of the main model management operators: Compose, Extract, Diff, Merge, Confluence, Domain, and Range. In this section, we describe Moda's mapping language and the semantics of its operators.

Currently, Moda supports mapping formulas expressed as *embedded dependencies* [1, p. 217], i.e., sentences of the form

$$\forall\mathbf{x}(\varphi(\mathbf{x}) \rightarrow \exists\mathbf{y}(\psi(\mathbf{x},\mathbf{y})))$$

where $\mathbf{x}$, $\mathbf{y}$ are lists of variables and φ, ψ are conjunctions of relational or equality atoms. Embedded dependencies include GLAV constraints of the form $Q_1 \subseteq Q_2$, where Q_1, Q_2 are select-project-join queries. In our implementation, dependencies are represented using data structures that mimic the parse trees of the expressions.

A *relational schema* is a tuple (σ, Σ), where $\sigma = \{\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n\}$ is a schema signature and Σ is a set of embedded dependencies over σ. Typically, Σ contains functional and inclusion dependencies over the signature. A *relational mapping* is a tuple (s_1, s_2, Σ_{12}), where s_1 and s_2 are relational schemas and Σ_{12} is a set of embedded dependencies over the signatures of s_1 and s_2. A set of dependencies is interpreted as a conjunction.

EXAMPLE 11 In the motivating example, the view s_v is given by the relational algebra expression

$$\pi_{\mathrm{C,N}}(\sigma_{\mathrm{V=\text{“ACME”}}}(s.\mathrm{O} \bowtie s.\mathrm{S})) = v.\mathrm{M}$$

In the prototype, it is represented as (s, v, Σ) where Σ is

$$\{\forall c\forall n\forall p(s.\mathrm{O}(c,n,p), s.\mathrm{S}(p,\text{“ACME”})) \rightarrow v.\mathrm{M}(c,n)),$$
$$\forall c\forall n(v.\mathrm{M}(c,n) \rightarrow \exists p(s.\mathrm{O}(c,n,p), s.\mathrm{S}(p,\text{“ACME”})))\} \quad \square$$

Let $\mathcal{L}_R$ be the language whose expressions are relational schemas and mappings as defined above. To implement the definitions of Section 4, we need to specify the state-based semantics of $\mathcal{L}_R$, which is defined as follows. An instance of relational schema $s = (\{\mathrm{R}_1, \ldots, \mathrm{R}_n\}, \Sigma)$ is a database $x = \langle r_1, \ldots, r_n\rangle$ such that r_i is a finite relation of type R_i and x satisfies every dependency in Σ. An instance of relational mapping (s_1, s_2, Σ_{12}) is a pair (x, y) of databases such that $x = \langle r_1, \ldots, r_n\rangle$ is an instance of s_1, $y = \langle t_1, \ldots, t_m\rangle$ is an instance of s_2, and the database $\langle r_1, \ldots, r_n, t_1, \ldots, t_m\rangle$ satisfies every dependency in Σ_{12}. That is, the mapping given by (s_1, s_2, Σ_{12}) is guaranteed to satisfy Σ_{12} and all schema constraints of s_1 and s_2.

Implementation of operators. To implement the operators for $\mathcal{L}_R$, we need algorithms that take $\mathcal{L}_R$ models and mappings as input and produce as output $\mathcal{L}_R$ models and mappings that satisfy the definitions of Section 4. It turns out that $\mathcal{L}_R$ is not closed under the model-management operators, i.e., some outputs may not be representable in $\mathcal{L}_R$, but only in some higher-order language, such as existential second-order logic (∃SO). In general, it is undecidable whether or not a given ∃SO-formula can be represented by an equivalent first-order formula [39]. Therefore, our algorithms need to exploit the properties of the operators and their inputs to compute the outputs under some sufficient conditions. Below we give a high-level overview of what it takes to implement the operators.

We start with the operator Compose. Let $map_{12} = (s_1, s_2, \Sigma_{12})$ and $map_{23} = (s_2, s_3, \Sigma_{23})$ be the input mappings, such that $s_2 = (\{\mathrm{R}_1, \ldots, \mathrm{R}_n\}, \Sigma_2)$. The $\mathcal{L}_R$-result of composition, if it exists, is given by (s_1, s_3, Σ_{13}) where Σ_{13} is equivalent to a ∃SO-formula

$$\exists\mathrm{R}_1 \ldots \exists\mathrm{R}_n(\Sigma_{12} \wedge \Sigma_2 \wedge \Sigma_{23})$$

We present a general algorithm for composing embedded dependencies in [28]. To sketch: First, we skolemize the dependencies as SO dependencies, as in [16]. Second, we eliminate the existential predicates of the intermediate schema, obtaining a deductive closure restricted to source and target predicates. Finally, we eliminate existential Skolem functions using a multi-step procedure.

The operators Domain and Range can be computed using a similar reduction algorithm [28]. To see that, notice that Domain(map_{12}) is given by $(\sigma_1, \Sigma^d_{12})$, where σ_1 is the schema signature of s_1 and Σ^d_{12} is an $\mathcal{L}_R$-reduction of the ∃SO-formula

$$\exists \mathrm{R}_1 \ldots \exists \mathrm{R}_n (\Sigma_{12} \wedge \Sigma_2)$$

If the domain and range constraints for mappings map_1 and map_2 are Σ^d_1, Σ^r_1 and Σ^d_2, Σ^r_2, respectively, then the result of Confluence is given by

$$(\Sigma^d_1 \vee \Sigma^r_1 \rightarrow \Sigma_1) \wedge (\Sigma^d_2 \vee \Sigma^r_2 \rightarrow \Sigma_2)$$

The above formula still needs to be translated to $\mathcal{L}_R$ if such a reduction exists. If map_1 and map_2 are both total and surjective then $\Sigma^d_1 \vee \Sigma^r_1$ and $\Sigma^d_2 \vee \Sigma^r_2$ evaluate to true, and confluence can be computed as a conjunction $\Sigma_1 \wedge \Sigma_2$ of mapping constraints.

The implementation of Invert is straightforward, since the mapping expressions of $\mathcal{L}_R$ are symmetric with respect to the "left" and "right" schema. That is, for $map = (s_1, s_2, \Sigma_{12})$ we obtain Invert(map) as (s_2, s_1, Σ_{12}). Implementing Invert for non-symmetric languages, such as SQL, is far more challenging.

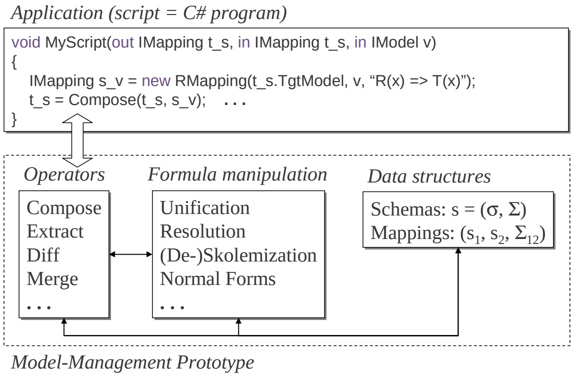
As we pointed out in Section 4, the outputs of the operators Extract, Diff, and Merge are not determined uniquely. The optimality criterion that drives our current implementation is based on the size of expressions used for schemas and mappings, i.e., it favors compact output schemas and mappings over more verbose ones.

Under these assumptions, operator Extract can be implemented using an algorithm similar to that of [27]. Let $\langle s_x, s\_s_x \rangle =$ Extract(s, map), where $s = (\sigma, \Sigma)$, $s_x = (\sigma_x, \Sigma_x)$. The output schema s_x is computed by copying into σ_x all relational symbols in σ that occur in the mapping map, and inferring all constraints from Σ that are relevant for σ_x. (Effectively, schema constraints are added to mapping constraints before invoking the operator.) For s_x constructed in this fashion, the mapping s_s_x is given as the view that populates each relation copied into s_x from its respective source relation in s. If map is a query on s, $\langle s_x, s\_s_x \rangle$ can be made more compact: s_x can be set to Range(map), and s_s_x can be set to map. However, determining that map is indeed a query, i.e., a functional mapping, requires a non-trivial analysis of the mapping expression [33].

Analogously to Extract, the operator Diff is computed using an algorithm that picks the relations not occurring in the input mapping. However, a more optimal solution can be obtained by exploiting the algorithms developed for computing relational view complements.

In a general case, the solution to Merge can be stated as a second-order formula, which we omit here for brevity. However, in some scenarios, such as data exchange [15], mappings are total and surjective, i.e., do not imply any within-schema constraints (such mappings are called conservative augmentations in [23]). In these cases, we can use a simpler strategy. Let $s_1 = (\sigma_1, \Sigma_1)$, $s_2 = (\sigma_2, \Sigma_2)$, and $map_{12} = (s_1, s_2, \Sigma_{12})$ be the inputs to the computation $\langle s, s\_s_1, s\_s_2 \rangle =$ Merge(s_1, s_1, map_{12}). Then, the output schema s is given as $s = (\sigma_1 \cup \sigma_2, \Sigma_1 \wedge \Sigma_2 \wedge \Sigma_{12})$, where $\sigma_1 \cup \sigma_2$ is the union of the schema signatures (with renaming upon name collisions). The mappings s_s_1 and s_s_2 are constructed as views on s as in the Extract implementation. Typically, the signature $\sigma_1 \cup \sigma_2$ contains substantial redundancy, which can be partially eliminated using equivalence-preserving schema transformations (e.g., those of [9, 32, 36]). Upon applying such transformations, the mappings s_s_1 and s_s_2 need to be adjusted accordingly and may become more complex, as illustrated in Example 4.

Architecture. The architecture of our prototype is depicted in Figure 8. The operators are invoked directly from a C# program and manipulate relational schemas and mappings as in-memory data structures. The implementation of the operators uses a library of building blocks that we developed for elementary formula manipulation. The library includes algorithms for unification, resolution, transforming a formula into implicative normal form, etc.



```
void MyScript(out IMapping t_s, in IMapping t_s, in IModel v)
{
   IMapping s_v = new RMapping(t_s.TgtModel, v, “R(x) => T(x)”);
   t_s = Compose(t_s, s_v);    . . .
}
```

Figure 8: Architecture of the prototype

Currently, the implementation comprises about 12000 lines of code, about half as much as Rondo's code base. It is relatively compact since it supports only the relational case, has no graphical user interface, and does not store schemas and mappings in a database system. A parser for mapping expressions is included.

We found that implementing the operators for $\mathcal{L}_R$ is both challenging and computationally expensive. Most notably, computing the composition, which seems to be the most frequently used operator, requires exponential worst-case time in the size of input mappings [28]. This is not surprising, given that many practical problems involving mappings are NP-hard (e.g., query optimization, query containment, and answering queries using views).

The overall running time of scripts largely depends on the complexity of the schema constraints and mapping expressions, as opposed to just their size. Because of that, it is difficult to present meaningful performance graphs. In our initial experience, scripts run in seconds only for quite simple mappings, such as the ones in Figure 2, and may require minutes for more complex ones. Developing more complete and more efficient algorithms is the subject of our ongoing work.

As a first application of Moda, we built a data migration tool to support schema evolution in a data-intensive application. The tool generates XSL transformations to migrate data from one version of the application to the next. The application uses several dozen schemas, which exploit inheritance, nesting, and other object-relational features. The $\mathcal{L}_R$-mapping between two subsequent versions of the application data is given by over 2500 embedded dependencies, which are translated into XSLT by chasing canonical instances under constraints. It takes slightly over six minutes to generate the final transformation on a 2.8 GHz PC. The XSLT produced as output comprises over 30K lines.

6. SCRIPT REWRITING

Model-management scripts have declarative semantics, as we explained in Section 3. However, just like relational algebra expressions, a script can be viewed as a specific execution plan. In fact, in Moda a script is given by a (procedural) C# program. Unsurprisingly, the execution order of the operators may affect the running time of the script. But more importantly, reordering the operators may turn a script that fails into an executable one:

EXAMPLE 12 Consider a model-management script

$$map = (m_1\_m_2 \circ m_2\_m_3) \circ m_3\_m_4$$

that is executed on the input mappings ($\forall$-quantifiers omitted)

$$\begin{aligned} m_1\_m_2 &= \ll \mathrm{E}(a,b) \rightarrow \exists v(\mathrm{F}(a,b), \mathrm{C}(a,v)) \gg \\ m_2\_m_3 &= \ll \mathrm{F}(a,b), \mathrm{C}(a,u), \mathrm{C}(b,v) \rightarrow \mathrm{D}(a,u,v) \gg \\ m_3\_m_4 &= \ll \mathrm{D}(a,u,v) \rightarrow \mathrm{H}(a,u) \gg \end{aligned}$$

between the models

$$\begin{aligned} m_1 &= \ll \mathrm{E}(\mathrm{A},\mathrm{B}) \gg & m_3 &= \ll \mathrm{D}(\mathrm{A},\mathrm{U},\mathrm{V}) \gg \\ m_2 &= \ll \mathrm{F}(\mathrm{A},\mathrm{B}), \mathrm{C}(\mathrm{A},\mathrm{V}) \gg & m_4 &= \ll \mathrm{H}(\mathrm{A},\mathrm{U}) \gg \end{aligned}$$

By exploiting [16, Theorem 3.6], it can be shown that the 3-colorability problem can be reduced to the mapping expression for $(m_1\_m_2 \circ m_2\_m_3)$. Therefore, the result of composing $m_1\_m_2$ and $m_2\_m_3$ is not expressible in first-order logic (let alone in $\mathcal{L}_R$). Hence, direct execution of the above script fails if the operators do not support second-order reasoning. However, by rewriting the script as

$$map = m_1\_m_2 \circ (m_2\_m_3 \circ m_3\_m_4)$$

we can effectively obtain the resulting mapping as

$$map = \ll \mathrm{E}(a,b), \mathrm{E}(b,c) \rightarrow \exists u(\mathrm{H}(a,u)) \gg \quad \square$$

Thus, sometimes a script cannot even be executed without first rewriting it as an equivalent script. In the example, we exploit associativity of composition to obtain an equivalent executable order of operators. Script rewriting can also exploit properties of models and mappings: in the motivating example, we know that s_v and $\mathsf{Invert}(v\_v')$ are queries that agree on their range, so we can replace the first three lines by the two lines shown below without changing the semantics of the script. This rewriting is similar to performing selection before join in relational query optimization:

1. $\langle v_d, v'\_v_d \rangle = \mathsf{Diff}(v', \mathsf{Invert}(v\_v'))$
2. $s\_v_d = s\_v \circ v\_v' \circ v'\_v_d$

Even in the simple example above, determining the equivalence of scripts is non-trivial and warrants a longer discussion, which is out of scope of this paper. Below we outline the key idea of our approach, which is based on equivalence-preserving transformations of operators. To enable such transformations, we introduce *minimality conditions* on the operators Merge, Extract, and Diff. To see why they are needed, notice that the output models in Definitions 2, 3, and 4 can be expanded by adding extra "irrelevant" states without violating the definitions. For example, if m is a relational schema produced by Merge, we can add an extra table to m and Definition 2 would still hold. The minimality conditions eliminate such irrelevant states and ensure that each output model is capable of representing just the needed information, and no other information.

For finite models, such as relational schemas with finite attribute domains, minimal models can be defined as those with the smallest number of instances. Using this minimality criterion, the definitions can be extended as follows: in Definition 2, m is a minimal model satisfying (i)-(iii); in Definition 3, m_x is a minimal model satisfying (i) and (ii); in Definition 4, m_d is minimal.

The minimality conditions allow us to establish important properties of the operators that are essential for script rewriting, such as commutativity and associativity, and tell us how to simplify the scripts when the mappings are functional or total [26]. Furthermore, we can relate the operators directly to some well-known problems studied in the database literature, as illustrated by the following proposition:

PROPOSITION 2 (VIEW COMPLEMENT) Let m_m_x be a total view from m onto m_x and let

$$\langle m_d, m\_m_d \rangle = \mathsf{Diff}(m, m\_m_x).$$

Then, m can be reconstructed from the views m_m_x and m_m_d. That is, the following holds:

$$\begin{aligned} &\langle m, m\_m_x, m\_m_d \rangle = \\ &\quad \mathsf{Merge}(m_x, m_d, \mathsf{Invert}(m\_m_x) \circ m\_m_d) \quad \square \end{aligned}$$

In general, if we merge models m_x and m_d obtained by Extract and Diff from some model m and mapping map, we get a (minimal) model m' that is isomorphic to m, and the isomorphism can be expressed using Composition and Confluence.

To study the properties of the operators, we utilize a special structure called an *instance graph*. Instance graphs are an analysis tool similar in spirit to canonical databases or rule-goal trees used for query analysis: properties of sets of models can be verified on a single representative structure. Specifically, an instance graph is a directed labeled graph whose nodes represent instances of models and edges denote pairs of instances that participate in mappings. For example, in the case of Merge the instance graph includes nodes for the two input models and the merged model.

In addition to being an analysis vehicle, instance graphs have an extra benefit as a data structure. To automate script rewriting, ultimately we would like to enumerate all useful "execution plans" for a script, such that the model-management system could pick the most efficient one – just as a database optimizer enumerates query execution plans. As a first step, we implemented a tool that attempts to find a counterexample that proves that two given scripts are not equivalent. The tool is similar in spirit to a satisfiability checker for logical formulas: it generates and checks different combinations of small, representative instance graphs. The randomized algorithm used in the tool is sound, but not complete, i.e., in some cases it may fail to find a counterexample. Finally, instance graphs could be used to define the minimality criterion for countably infinite models using homomorphisms between instance graphs.

Despite the usefulness of the minimality conditions for script rewriting, it is often impractical to enforce them on the execution of a specific operator sequence. In fact, for inputs given in $\mathcal{L}_R$, the minimal outputs of Merge, Extract, and Diff are often not expressible in $\mathcal{L}_R$ at all, are too hard to compute, or have an unnecessarily verbose representation. Consider the following example:

EXAMPLE 13 (Based on [19], Example 3.6): Suppose that attribute A has a finite integer domain $\mathcal{I}$, and let

$$\begin{aligned} &m = \ll \mathrm{R}(\mathrm{A},\mathrm{B}) \gg, \; m' = \ll \mathrm{T}(\mathrm{A}) \gg \\ &m\_m' = \ll \pi_{\mathrm{A}}(\mathrm{R}) = \mathrm{T} \gg \end{aligned}$$

Then, the minimal result for $\langle m_d, m\_m_d \rangle = \mathsf{Diff}(m, m\_m')$ is given by the minimal view complement

$$\begin{aligned} m_d &= \ll \mathrm{S}_1(\mathrm{B}), \ldots, \mathrm{S}_{|\mathcal{I}|}(\mathrm{B}) \gg \\ m\_m_d &= \ll \mathrm{S}_i = \pi_{\mathrm{B}}(\sigma_{\mathrm{A}=i}(\mathrm{R})) \text{ for } i \in \mathcal{I} \gg \end{aligned}$$

If $\mathcal{I}$ is the domain of positive 32-bit integers, then the output schema would contain over two billion table definitions. Although the schema is minimal in the information-theoretic sense, it is by no means syntactically compact. A non-minimal, yet probably more useful complement is obtained as $\mathrm{S} = \mathrm{R}$. $\square$

For the reasons outlined above, our implementation does not guarantee minimal output models, but makes an effort to produce outputs that are close to the minimal ones. Minimality can be compared to numerical precision: equivalence-preserving rewriting of arithmetic operations on real numbers helps speed up numeric calculation; however, the actual computation of the rewritten formula is typically performed on approximations given by floating point numbers.

7. RELATED WORK

Nearly all previous work on model management has considered the behavior of operators on some representations of models and mappings, not on data instances related by mappings [6, 7, 8, 27, 30]. One exception is [3], where the operators are defined axiomatically in terms of other operators using category theory. Another is the recent work [16], which uses the state-based approach to characterize mapping composition. Uniform representation of heterogeneous schema and instance data is discussed in [14].

In [27], we suggested simple correctness conditions that characterize the information capacity of the schemas returned by the operators. Specifically, we required that the schema produced by Merge be at least as expressive as each of the input schemas, and the schema delivered by Extract be no more expressive than the input schema. However, these requirements are quite weak and do not specify what the mappings produced by the operators should do.

In this paper, we give operator definitions that do work on executable mappings and satisfy the desiderata of well known, but disparate problems studied in the literature, such as integration of views [9, 12, 36] and of schemas [3, 10, 20, 22, 30], composition of queries [34] and of schema mappings [16, 24], view complement [4, 19], view selection [2, 13, 21, 38], and answering queries using views [11, 17]. These problems have typically been studied in isolation and trimmed to specific languages. We distill essential properties of these problems into language-independent operators.

Operator Compose generalizes query composition. It is equivalent to query composition when $m_1\_m_2$ and $m_2\_m_3$ are queries, i.e., functional mappings. It is well-known how to compose two relational algebra queries, and that the result is always another relational algebra query [1]. However, if the input mappings are non-functional or have different directionality, computing composition can be much harder. A first general definition of mapping composition was given in [24], where the result of composition is defined relative to a query language. The language-independent definition used in this paper was proposed independently in [16] and [25]. The work [16] was the first to demonstrate how to turn a state-based operator definition into algorithms that manipulate executable mappings. Among the negative results, [16] proves that composing mappings given by source-to-target tuple-generating dependencies (tgds) may not be definable by any formula of $L^{\omega}_{\infty\omega}$ logic, and that composing first-order formulas may produce a non-computable mapping. Work [28] shows that composition of full tgds that need not be source-to-target is undecidable. On the positive side, [16] establishes that full source-to-target tgds and second-order tgds are closed under composition and are suitable mapping languages for data exchange [15].

Our formalization of Merge builds on the extensive work on schema and view integration. To our knowledge, Definition 2 is the first to satisfy the following important desiderata suggested in that literature. First, the definition is language independent [3, 30]. Second, Merge is driven by an input mapping, and the output includes the mappings between the merged model and the input models [9, 22, 36]. Third, the merged model represents the complete information of each input model [12, 30, 36]. Fourth, inconsistent models can be merged [22]. Finally, Merge is associative and commutative, satisfying the desiderata of [10].

Our definition of Extract builds on the (materialized) view selection problem [2, 13, 21, 38], whose objective is to find a set of views that allows answering a given query workload. If the workload consists of a single query map, the correctness criterion of view selection is condition (i). Past work on view selection is an example of where the focus has been on finding the *optimal* representation of the result of Extract, where the language for expressing models and mappings are various subsets of SQL. Condition (i) can also be interpreted as a problem of answering queries using views using an exact rewriting [11, 17]. That is, given a view m_m_x, the goal is to rewrite query map on m into query $q = \mathsf{Invert}(m\_m_x) \circ map$ on m_x. Definition 3 covers a general case in which map is an arbitrary, possibly non-functional mapping. Algorithms for query reachability (e.g., [18, 37]) can be used to implement Extract when the mapping language is recursive datalog (and subsets thereof). The extracted schema is the result of looking at the leaves of the query tree after the predicates in the query have been applied as tightly as possible to all nodes in the tree. An implementation of Extract for SQL can exploit view selection algorithms that are deployed in commercial database systems [2].

Our definition of Diff is based on the view complement problem [4, 19], yet covers the general case in which the input mapping is non-functional (i.e., not a view). Two views are complementary if given the state of each view, there is a unique corresponding state of the source database. That is, if the two views are materialized then the database can be reconstructed from the views. View complements are exploited to guarantee desirable data warehouse properties such as independence and self-maintainability. The polynomial algorithm of [19] can be used for computing Diff when the input map is a relational select-join view. If map contains projections, the output view may be sensitive to permutation of constants.

The minimality conditions that we discussed in the context of script rewriting have been suggested for view minimization, finding minimal view complements, or minimizing the output schemas in view integration. Although they are essential for performing equivalence-preserving transformations, many authors argued for approximate solutions to these individual problems. For example, [21] selects views that are minimal relative to a given set of views, but not w.r.t. all conceivable views (i.e., non-minimal Extract is acceptable); [19] argues that the reduced information content of minimal (vs. non-minimal) view complements may not justify the increase in their complexity (i.e., non-minimal Diff is acceptable); [12] describes an algorithm for minimizing the merged schema but does not guarantee a minimal result due to complexity caused by schema constraints. Thus, we use the minimality criteria as a foundation for algebraic optimization, but adopt a best-effort policy for our implementation.

In [29], morphisms are referred to as inter-schema correspondences and are used to generate executable mappings by exhaustive enumeration of the possible interpretations.

8. CONCLUSIONS

We presented a state-based approach to specifying the semantics of model-management operators, which is instrumental for building model-management systems that work with executable mappings. A major strength of state-based characterization is its ability to specify the operators in an abstract fashion, without appealing to particular schema, constraint, or transformation languages, or to particular representations of models and mappings. We showed that our semantics satisfy the desiderata of well known, but disparate problems studied in the literature. We believe that our approach lays a foundation for a formal treatment of many model-management problems.

We explored two paths to implementing the operators. First, we leveraged an existing implementation by generating executable mappings from the output schemas and morphisms produced by the Rondo system. To do that, we identified the language $\mathcal{L}_P$ of tree schemas and path-morphisms for which Rondo's outputs respect our state-based semantics. Second, to overcome the limitations of path-morphisms, we built a new prototype Moda that implements the model-management operators for the language $\mathcal{L}_R$ of relational schemas and mappings. We found that representing schemas and mappings using explicit logical formulas is a more flexible, yet substantially more challenging approach, which raises deep theoretical issues. Third, we identified the potential of script rewriting, which is critical not only for performance reasons but also to ensure that the script can be executed at all. Finally, we applied Moda to build a schema evolution tool for a data-intensive application.

Further work is needed to find other interpretations of morphisms to cover more expressive mappings, to develop efficient algorithms for implementing model-management operators where mappings are expressed as logic formulas, and to obtain a deeper understanding of the algebraic and computational properties of the operators.

Acknowledgements The authors thank Alan Nash for helpful discussions and comments on a preliminary version of this paper.

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, Mass., 1995.

[2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated Selection of Materialized Views and Indexes in Microsoft SQL Server. In *Proc. VLDB*, 2000.

[3] S. Alagic and P. A. Bernstein. A Model Theory for Generic Schema Management. In *Proc. DBPL*, 2001.

[4] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM TODS*, 6(4):557–575, 1981.

[5] BEA, IBM, Microsoft, SAP, and Siebel. Business Process Execution Language for Web Services Version 1.1, 2003.

[6] P. A. Bernstein. Applying Model Management to Classical Metadata Problems. In *Proc. CIDR*, 2003.

[7] P. A. Bernstein, A. Y. Halevy, and R. Pottinger. A Vision of Management of Complex Models. *SIGMOD Record*, 29(4):55–63, 2000.

[8] P. A. Bernstein and E. Rahm. Data Warehouse Scenarios for Model Management. In *Proc. ER*, pages 1–15, Oct. 2000.

[9] J. Biskup and B. Convent. A Formal View Integration Method. In *Proc. ACM SIGMOD*, pages 398–407, 1986.

[10] P. Buneman, S. B. Davidson, and A. Kosky. Theoretical Aspects of Schema Merging. In *Proc. EDBT*, 1992.

[11] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. View-based Query Processing: On the Relationship between Rewriting, Answering, and Losslessness. In *Proc. ICDT*, 2005.

[12] M. A. Casanova and V. M. P. Vidal. Towards a Sound View Integration Methodology. In *Proc. PODS*, 1983.

[13] R. Chirkova, A. Y. Halevy, and D. Suciu. A Formal Perspective on the View Selection Problem. In *Proc. VLDB*, pages 59–68, 2001.

[14] L. Delcambre and D. Maier. Models for Superimposed Information. In *Workshop on Conceptual Models for the WWW in conjunction with ER Conf.*, pages 264–280, 1999.

[15] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. In *Proc. ICDT*, pages 207–224, 2003.

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

[17] A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 10(4):270–294, 2001.

[18] A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Static Analysis in Datalog Extensions. *Journal of the ACM*, 48(5):971–1012, 2001.

[19] J. Lechtenbörger and G. Vossen. On the Computation of Relational View Complements. *ACM TODS*, 28(2):175–208, 2003.

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

[21] C. Li, M. Bawa, and J. D. Ullman. Minimizing View Sets without Loosing Query-Answering Power. In *Proc. ICDT*, pages 99–103, 2001.

[22] J. Lin and A. O. Mendelzon. Merging Databases Under Constraints. *Intl. Journal of Cooperative Information Systems*, 7(1):55–76, 1998.

[23] J. Madhavan, P. A. Bernstein, P. Domingos, and A. Y. Halevy. Representing and Reasoning about Mappings between Domain Models. In *AAAI/IAAI 2002*.

[24] J. Madhavan and A. Halevy. Composing Mappings Among Data Sources. In *Proc. VLDB*, 2003.

[25] S. Melnik. *Generic Model Management: Concepts and Algorithms*. Ph.D. thesis, University of Leipzig, Springer LNCS 2967, 2004.

[26] S. Melnik, P. A. Bernstein, A. Halevy, and E. Rahm. A Semantics for Model Management Operators. Technical Report MSR-TR-2004-59, Microsoft Research, 2004.

[27] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *Proc. ACM SIGMOD*, 2003.

[28] A. Nash, P. A. Bernstein, and S. Melnik. Composing Mappings Given by Embedded Dependencies. In *Proc. PODS*, 2005.

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

[30] R. Pottinger and P. A. Bernstein. Merging Models Based on Given Correspondences. In *Proc. VLDB*, 2003.

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

[32] A. Rosenthal and D. S. Reiner. Tools and Transformations – Rigorous and Otherwise – for Practical Database Design. *ACM TODS*, 19(2):167–211, 1994.

[33] L. Segoufin and V. Vianu. Views and Queries: Determinacy and Rewriting. In *Proc. PODS*, 2005.

[34] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *Proc. VLDB*, pages 261–270, 2001.

[35] J. Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996.

[36] S. Spaccapietra and C. Parent. View Integration: A Step Forward in Solving Structural Conflicts. *TKDE*, 6(2):258–274, 1994.

[37] D. Srivastava and R. Ramakrishnan. Pushing Constraint Selections. *Journal of Logic Programming*, 16(3–4):361–414, 1993.

[38] D. Theodoratos, S. Ligoudistianos, and T. K. Sellis. View Selection for Designing the Global Data Warehouse. *Data and Knowledge Engineering (DKE)*, 39(3):219–240, 2001.

[39] J. van Benthem and K. Doets. Higher-order logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. 1*. Reidel, Dordrecht, 1983.