A Semantics for Model Management Operators

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, Extract, and Merge, that are applied to models, such as database schemas or interface specifications, and to mappings between models. To be used in practice, these operators need to be implemented for particular schema definition languages and mapping languages. To guide that implementation, we need a language-independent semantics that tells what the operators should do.

In this paper we develop a state-based semantics of the operators. That is, we express the effect of applying the operators to models in terms of what the operators do to instances of these models. We show that our semantics captures previously-proposed desiderata for the operators. We study formal properties of the operators, such as commutativity, associativity, uniqueness of results, and how the cardinality of the results corresponds to that of the inputs. Finally, we specify the state-based semantics of the operators in Rondo, the first prototype model-management system.

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 domains.

Model management aims at providing a generic and powerful environment to enable rapid development of metadata-intensive applications in different domains [6, 7]. 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, Merge, Extract, Diff, and Compose. These operators can be used for solving schema evolution, data integration, and other scenarios using short programs, or *scripts* [6, 8], which are executed by a model management system. The first prototype of such a system, called Rondo, was presented in [25].

Rondo has a precise semantics for each operator's effect on models and mappings. But this is not the whole story. Most model-management scripts generate mappings for transforming instances of models, e.g., scripts for data or message translation or for wrapping a database. 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 specifications are 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. We first specify a generic semantics. Then, to show its utility, we use it to specify the semantics of Rondo.

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. We use the terms instance and state interchangeably.

Our second contribution is showing that our semantics satisfy the desiderata of well known, but disparate problems studied in the literature, such as integration of views [9, 12, 32] and of schemas [3, 10, 19, 21, 28], composition of queries [30] and of schema mappings [23], view complement [4, 18], view selection [2, 13, 20, 34], and answering queries using views [11, 15]. These problems have typically been studied in isolation and trimmed to specific languages. We distill essential properties of these problems into our set of language-independent operators.

Our third contribution is to study several formal properties of model-management operators. Here we face the following technical challenge. The definitions of most of the operators involve a condition on *minimality* of the resulting model. The minimality condition is, in essence, a second-order condition on models, whose properties are hard to study. We show that it is possible to provide an equivalent characterization of the operator that involves only a first-order condition on models. Given that characterization, we can verify certain properties of operators on a structure called *instance graph*, which provides a canonical representation of a set of models (similar in spirit to the use of canonical databases for checking query containment).

We use the above techniques to prove several results: We show that the result of some operators (e.g., Compose, Confluence) is unique, while for others (e.g., Extract, Merge) it is unique up to isomorphism. We show the relationship between the cardinalities of the inputs and outputs of the operators. And we examine some basic properties such as commutativity and associativity of operators.

Our final contribution is to use state-based semantics to specify the behavior of operators in Rondo [25]. Rondo's operators are defined for morphisms, a simple mapping language whose expressions are sets of arcs connecting the elements of two schemas. We define the semantics of a useful subset of this language, called path-morphisms. We show that Rondo operators satisfy the state-based semantics of model management operators with respect to path-morphisms. Thus, every Rondo script produces valid path morphisms whose semantics can be easily understood.

Nearly all previous work on model management has considered the behavior of operators only on models and mappings, not on data instances related by mappings [6, 7, 8, 25, 28]. The one exception we know of is [3], where the operators are defined axiomatically in terms of other operators using category-theory. For example, schema integration, which underlies the operator Merge, is defined using a categorical construct called pushout. The approach is developed within an abstract category of schemas and has only been applied to issues of constraint preservation.

The paper is organized as follows. Section 2 formally defines models, mappings, and operators. Section 3 defines the semantics of the operators and studies their properties. Section 4 uses state-based semantics to characterize the behavior of the Rondo prototype. Section 5 is the conclusion.

2 Problem definition

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 [31]. 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 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 $\ll$E$\gg$ for expression E.

EXAMPLE 1 Each instance of $m = \ll\mathrm{R(A,B)},\ \mathrm{S(C)}\gg$ is an entire populated relational database. If $|\mathrm{A}|$, $|\mathrm{B}|$ and $|\mathrm{C}|$ are domain sizes of the respective attributes, then model m has $2^{|\mathrm{A}|\cdot|\mathrm{B}|+|\mathrm{C}|}$ instances, one of which is the empty database. $\square$

Our definitions will often refer to models that are *minimal* w.r.t. a set of models $\mathcal{M}$. A model m' is *minimal* w.r.t. $\mathcal{M}$ if it has the smallest cardinality of all models in $\mathcal{M}$, i.e., $m' \in \mathcal{M}$ and $\forall m \in \mathcal{M} : m' \preceq m$, where $m' \preceq m$ iff there exists a surjective function from m onto m'.

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. To refer to mappings, we use variables map_1, map_2, $m_1\_m_2$, $m_2\_m_3$, etc.

EXAMPLE 2 Consider two relational schemas

$$m_1 = \ll\mathrm{R}(\underline{\mathrm{ID}}, \mathrm{Age})\gg, \quad m_2 = \ll\mathrm{S}(\underline{\mathrm{ID}}, \mathrm{Sex})\gg.$$

The mapping $m_1\_m_2 = \ll\pi_{\mathrm{ID}}(\mathrm{R}) = \pi_{\mathrm{ID}}(\mathrm{S})\gg$ is a binary relation on the instances of m_1 and m_2. That is, for all $x \in m_1$, $y \in m_2$: $(x, y) \in m_1\_m_2$ iff $\pi_{\mathrm{ID}}(x.\mathrm{R}) = \pi_{\mathrm{ID}}(y.\mathrm{S})$. $\square$

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 exist at the same time 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]).

A mapping can be thought of as a constraint that holds between two models [9, 21, 22]. 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. Models 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. A *query* is a functional mapping. A query (and hence a mapping in general) may not be expressible in a specific query language [1].

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. In this paper we consider the operators Match, Compose ($\circ$), Merge, Extract, Diff, and Confluence ($\oplus$).

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. Given values for the input variables, an *exact answer* to a script consists of the values of the output variables in a valid substitution. Two scripts are *equivalent* if they return the same exact answers for every given set of inputs.

EXAMPLE 3 The script shown below integrates the "overlapping" portions of models m_1 and m_2 based on the mapping $m_1\_m_2$ (the individual operators are defined formally in Section 3 and are used here only to illustrate how they can be combined into scripts). Intuitively, the script extracts the portions p of m_1 and q of m_2 that "participate" in $m_1\_m_2$, and merges them into a model m. The outputs are model m and mappings m_m_1, m_m_2 between m and the input models:

script $\mathsf{Intersect}(m_1, m_2, m_1\_m_2)$
$\langle p, m_1\_p\rangle = \mathsf{Extract}(m_1, m_1\_m_2)$;
$\langle q, m_2\_q\rangle = \mathsf{Extract}(m_2, \mathsf{Invert}(m_1\_m_2))$;
$\langle m, m\_p, m\_q\rangle =$
$\mathsf{Merge}(p, q, (\mathsf{Invert}(m_1\_p) \circ m_1\_m_2) \circ m_2\_q)$;
$m\_m_1 = m\_p \circ \mathsf{Invert}(m_1\_p)$;
$m\_m_2 = m\_q \circ \mathsf{Invert}(m_2\_q)$;
return $\langle m, m\_m_1, m\_m_2\rangle$ $\square$

The script is a conjunction of expressions delimited by semicolons. Computing the script for the inputs of Example 2 produces a substitution of the output variables, such as $m = \ll\mathrm{T}(\underline{\mathrm{ID}})\gg$, $m\_m_1 = \ll\mathrm{T} = \pi_{\mathrm{ID}}(\mathrm{R})\gg$, $m\_m_2 = \ll\mathrm{T} = \pi_{\mathrm{ID}}(\mathrm{S})\gg$. $\square$

Problem statement: Ultimately, our goal is to build model-management systems that can be deployed in practical settings and execute scripts efficiently for complex model and mapping languages. In support of this goal, we focus on the following problems in this paper.

First, we develop state-based characterizations of model-management operators. Ideally, operator specifications should be language-independent, yet compatible with the problems examined in the literature for specific languages. Second, we explore whether the result of an operator is unique (or at least up to isomorphism) to establish invariants that are guaranteed to hold across different implementations of operators. Third, since computing the results of operators can be expensive, optimization is important. For example, we may want to rewrite a script into an equivalent script that can be executed more efficiently. Hence, we study properties of operators that provide the foundation for optimizations. To illustrate, if Compose ($\circ$) is associative, then we can replace the third parameter to Merge in Example 3 by $\mathsf{Invert}(m_1\_p) \circ (m_1\_m_2 \circ m_2\_q)$.

Given the operator definitions and a particular model and mapping language, we can then define more specific properties, such as the following:

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 said to be* closed under θ, *if given any inputs to θ in $\mathcal{L}$, the output can also be expressed in $\mathcal{L}$.* $\square$

If $\mathcal{L}$ is closed under θ, then we can ask whether the *representation* of the result of θ is unique, and if not, whether we can find the most *computationally efficient* representation. Computational efficiency is affected by various criteria. For example, in view selection (which we relate to our Extract operator) the criteria include query evaluation cost or size of schema instances [2, 20, 34]. In view integration (which we relate to Merge), the criteria involve syntactic properties, such as size of expressions used for schemas and queries [9, 32].

Throughout this paper we show how a wide range of previous work can be viewed as studying these properties for particular languages. And in Section 4 we study the language supported by our prototype Rondo.

3 Design and analysis of operators

In this section we suggest a state-based semantics for the key operators proposed in the literature [6, 7, 8, 25]. We discuss six major operators: Match, Compose, Merge, Extract, Diff, and Confluence, and five auxiliary operators: cross-product, Id, Invert, Domain, and Range. Using these operators, we have been able to characterize all model-management tasks that have appeared in the literature. However, whether the operators are complete or best is an open question.

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 [29] but ultimately Match depends on human feedback (and hence may be partial or even inaccurate).

The auxiliary operators are defined as follows:

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

- $\mathsf{Id}(m) =_{\text{df}} \{(x, x) \mid x \in m\}$

The above operators have standard algebraic definitions. Thus, many well-known properties hold, such as $\mathsf{Domain}(\mathsf{Id}(m)) = m$ or $\mathsf{Invert}(\mathsf{Invert}(map)) = map$.

3.1 Compose operator

To motivate the definition of the Compose operator, consider the following example.

EXAMPLE 4 Let $m_1\_m_2$ be an export mapping that generates an XML document $y \in m_2$ from a relational database $x \in m_1$. Suppose XML schema m_2 is modified into schema m_3. Let $m_2\_m_3$ be the mapping between the original and the new XML schema. To derive the updated export mapping, we compute the composition of $m_1\_m_2$ and $m_2\_m_3$, denoted as $\mathsf{Compose}(m_1\_m_2, m_2\_m_3)$ or simply as $m_1\_m_2 \circ m_2\_m_3$. □

The following definition describes formally the properties of the updated mapping in the above example. It is consistent with mapping composition scenarios studied in the literature [8, 23, 30]:

DEFINITION 2 (COMPOSE, $\circ$)

$$m_1\_m_2 \circ m_2\_m_3 =_{\text{df}}$$
$$\{(x, z) \mid (x, y) \in m_1\_m_2 \text{ and } (y, z) \in m_2\_m_3\} \quad \square$$

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. This case is illustrated in the following example.

EXAMPLE 5 Let

m_1 = ≪R(A,B)≫, m_2 = ≪S(A,B)≫, m_3 = ≪T(B)≫
$m_1\_m_2$ = ≪$\mathrm{S} = \sigma_{\mathrm{A}>5}(\mathrm{R})$≫
$m_2\_m_3$ = ≪$\mathrm{T} = \pi_{\mathrm{B}}(\mathrm{S})$≫

Then, the result of composition can be specified as

$m_1\_m_2 \circ m_2\_m_3$ = ≪$\mathrm{T} = \pi_{\mathrm{B}}(\sigma_{\mathrm{A}>5}(\mathrm{R}))$≫ □

Many well-known properties hold on Compose, including the following ones:

PROPOSITION 1

1. Compose is associative, i.e., $map_1 \circ (map_2 \circ map_3) = (map_1 \circ map_2) \circ map_3$
2. Compose is not commutative. Instead: $map_1 \circ map_2 = \mathsf{Invert}(\mathsf{Invert}(map_2) \circ \mathsf{Invert}(map_1))$
3. Mapping map is a surjective function onto m if and only if $\mathsf{Invert}(map) \circ map = \mathsf{Id}(m)$ □

Computing the results of composition for concrete languages can be very hard. For example, [23] shows that the GLAV mapping language is *not* closed under composition, and studies the complexity of computing composition in certain cases where it is possible. The work [30] presents algorithms for composing an XML publishing view with an XQuery and decomposing the result into a SQL query and a tagging graph. It shows that specialized languages may have to be developed when the result of composition is not representable in any existing language.

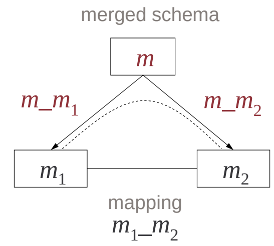

Figure 1: Illustration of Merge (Example 6)

3.2 Merge operator

We explain the intuition behind the Merge operator using the following view integration scenario.

EXAMPLE 6 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 1). 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 and no other information, i.e., is minimal. 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.

DEFINITION 3 (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 if and only if*

i. *m_m_1 and m_m_2 are surjective functions onto m_1 and m_2, respectively*

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

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

iv. *m is a minimal model satisfying (i)-(iii).* □

Condition (i) states that m_m_1 and m_m_2 are (possibly partial) 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 all information in m is useful, for either view m_m_1 or view m_m_2. The minimality condition (iv) prevents m from containing extra information that is not necessary for representing all of m_1 and m_2.

EXAMPLE 7 Let

$m_1 = \ll\mathrm{R}(\underline{\mathrm{ID}}, \mathrm{Age})\gg$
$m_2 = \ll\mathrm{S}(\underline{\mathrm{ID}}, \mathrm{Sex})\gg$
$m_1\_m_2 = \ll\pi_{\mathrm{ID}}(\mathrm{R}) = \pi_{\mathrm{ID}}(\mathrm{S})\gg$

Then, the result of Merge can be specified as

$m = \ll\mathrm{T}(\underline{\mathrm{ID}}, \mathrm{Age}, \mathrm{Sex})\gg$
$m\_m_1 = \ll\mathrm{R} = \pi_{\mathrm{ID,Age}}(\mathrm{T})\gg$
$m\_m_2 = \ll\mathrm{S} = \pi_{\mathrm{ID,Sex}}(\mathrm{T})\gg$

Conditions (i)-(iii) of Definition 3 are easy to verify. The minimality of the merged model can be tested with help of Lemma 1 that we present below. □

Our formalization of Merge builds on the extensive work on schema and view integration. To our knowledge, Definition 3 is the first to satisfy the following important desiderata suggested in that literature. First, the definition is language independent [3, 28]. Second, Merge is driven by an input mapping, and the output includes the mappings between the merged model and the input models [9, 21, 32]. Third, the merged model represents the complete information of each input model [12, 28, 32]. Fourth, inconsistent models can be merged [21] (see Corollary 1). This case may happen when one of the models may be in a state that corresponds to no valid state of another model, i.e., $m_1\_m_2$ is not a total surjective mapping. Finally, Theorem 1 shows that Merge is associative and commutative [10]. It states the main results of this section.

THEOREM 1 Merge has the following properties:

1A. The output model m and mappings m_m_1, m_m_2 of Merge are determined up to isomorphism.

1B. If $\langle m, m\_m_1, m\_m_2\rangle = \mathsf{Merge}(m_1, m_2, m_1\_m_2)$ then $|m| = |m_1\_m_2| + |m_1 - \mathsf{Domain}(m_1\_m_2)| + |m_2 - \mathsf{Range}(m_1\_m_2)|$

1C. Merge is commutative. That is:

$\langle m, m\_m_1, m\_m_2\rangle = \mathsf{Merge}(m_1, m_2, m_1\_m_2)$

if and only if $\langle m, m\_m_2, m\_m_1\rangle =$ $\mathsf{Merge}(m_2, m_1, \mathsf{Invert}(m_1\_m_2))$.

1D. Merge is associative, i.e., the following scripts 3wayM1 and 3wayM2 are equivalent:

script 3wayM1($m_1, m_2, m_3, m_1\_m_2, m_2\_m_3$)
 $\langle m_{12}, m_{12}\_m_1, m_{12}\_m_2\rangle =$
  $\mathsf{Merge}(m_1, m_2, m_1\_m_2)$;
 $\langle m, m\_m_{12}, m\_m_3\rangle =$
  $\mathsf{Merge}(m_{12}, m_3, m_{12}\_m_2 \circ m_2\_m_3)$;
return $\langle m, m\_m_{12} \circ m_{12}\_m_1,$
  $m\_m_{12} \circ m_{12}\_m_2, m\_m_3\rangle$

script 3wayM2($m_1, m_2, m_3, m_1\_m_2, m_2\_m_3$)
 $\langle m_{23}, m_{23}\_m_2, m_{23}\_m_3\rangle =$
  $\mathsf{Merge}(m_2, m_3, m_2\_m_3)$;
 $\langle m, m\_m_{23}, m\_m_1\rangle =$
  $\mathsf{Merge}(m_{23}, m_1, m_{23}\_m_2 \circ \mathsf{Invert}(m_1\_m_2))$;
return $\langle m, m\_m_1, m\_m_{23} \circ m_{23}\_m_2,$
  $m\_m_{23} \circ m_{23}\_m_3\rangle$ □

Part 1A tells us that although the output model and mappings of Merge are not determined uniquely (e.g., in contrast to Compose), they are guaranteed to capture the same amount of information. The cardinality $|m|$, or information capacity [17, 26], of the output model m is given explicitly in 1B. If the cardinalities are infinite, m consists of three disjoint partitions whose cardinalities are specified by the summands.

Before discussing the proof of Theorem 1, we point out the following corollary of Part 1B of the theorem.

COROLLARY 1 Let $\langle m, m\_m_1, m\_m_2\rangle = \mathsf{Merge}(m_1, m_2, m_1\_m_2)$.

1. If m_1 and m_2 are consistent under $m_1\_m_2$, then $|m| = |m_1\_m_2|$
2. If $m_1\_m_2 = m_1 \times m_2$, then $|m| = |m_1| \cdot |m_2|$
3. If $m_1\_m_2 = \emptyset$, then $|m| = |m_1| + |m_2|$ □

The corollary illustrates that (in contrast to the definition suggested in [21]) Merge does not lose information even under contradictory integration constraints, i.e., when $m_1\_m_2 = \emptyset$.

PROOF SKETCH: The proof of Theorem 1 proceeds in two steps. The first step addresses the technical difficulty of dealing with condition (iv) in Definition 3. The condition requires that the result be a minimal model, thereby specifying a condition on *all* models. The following lemma provides an alternative but equivalent characterization of Merge, where condition (iv) is replaced by a condition that can be checked on a *single* candidate model. In the lemma, $map[x]$ denotes the projection of x over map, defined as $map[x] =_{\mathrm{df}} \{y \mid (x, y) \in map\}$.

LEMMA 1 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 if and only if

1. Conditions (i)-(iii) of Definition 3 hold, and
2. For all $z_1, z_2 \in m$: if $m\_m_1[z_1] = m\_m_1[z_2]$ and $m\_m_2[z_1] = m\_m_2[z_2]$ then $z_1 = z_2$ □

The lemma provides additional insight into the properties of Merge. Each pair $(x, y) \in m_1\_m_2$ corresponds to a single valid state z of the merged model m. Upon merging, two mutually consistent states x and y are "glued" into z in such a way that we can unambiguously reconstruct x and y from z using two functional mappings[1], m_m_1 and m_m_2. Using condition (2) of Lemma 1, we can verify immediately that model m in Example 7 is minimal. To prove that (2) is necessary for minimality of Merge, we show that assuming the opposite allows us to construct a smaller model m' that satisfies (i)-(iii), leading to a contradiction. To show that (2) is sufficient we establish a lower bound k on the number of instances in m using (i)-(iii) and demonstrate that each model that satisfies the lemma has exactly k instances, i.e., is minimal.

Now that we can verify locally that a model and a pair of mappings are actually a merge, we can study the properties

[1] Since m_m_1 and m_m_2 are functions, $m\_m_1[x]$ and $m\_m_2[x]$ are singleton sets. $map[x]$ is used in its general form in Lemmas 2 and 3.

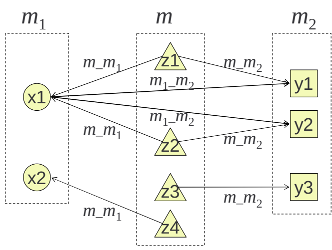

Figure 2: Instance graph illustrating Merge

of Merge on 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.

An instance graph is a directed labeled graph whose nodes represent instances of models and edges denote pairs of instances that participate in mappings. A pair $(x, y) \in map$ is represented as a directed edge from x to y labeled with map. The direction of edges is used to distinguish between the "left" and "right" instances of a mapping and does not imply that the mapping is functional. In the case of Merge the instance graph includes nodes for the two input models and the merged model. Figure 2 depicts an instance graph that shows the result of merging models $m_1 = \{x_1, x_2\}$ and $m_2 = \{y_1, y_2, y_3\}$. The boundaries of models are marked using dashed boxes. The instance graph illustrates that m contains exactly one instance for each edge of $m_1\_m_2$ and for each instance of m_1 and m_2 that is not incident on $m_1\_m_2$.

With the help of instance graphs, the proof of Theorem 1 proceeds as follows. The proof of Part 1A is that two graphs with input models and mappings that satisfy Lemma 1 must be isomorphic. Part 1B can be demonstrated by splitting the instances of m_1 and m_2 into three disjoint partitions based on whether or not they participate in $m_1\_m_2$. The proof of commutativity follows from the definition of Merge, while associativity can be sketched using instance graphs as follows: we traverse the instances of m_2 that are connected to those of m_1 and m_3 in the instance graph that represents a 3-way merge, and show that their "projections" over Invert$(m_1_m_2)$ and $m_2\_m_3$ are equivalent in 3wayM1 and 3wayM2. Each unconnected instance of m_1, m_2, and m_3 has exactly one counterpart in m. □

To conclude this section, we note that [9] describes an algorithm for implementing Merge for relational schemas under the condition that m_1 and m_2 are consistent (or "conflict-free") under $m_1\_m_2$. On the flip side, the authors note that this condition is undecidable for the considered constraint language, which uses functional, inclusion, and exclusion dependencies.

3.3 Extract operator

The operator Extract takes a model m and a mapping map between m and some model m', and returns a portion m_x of m that participates in the mapping. We begin with a motivating example.

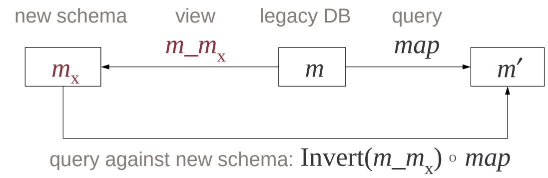

Figure 3: Illustration of Extract (Example 8)

EXAMPLE 8 Let m be a legacy database schema and map be a query over m. Our goal is to upgrade the legacy database by producing a new schema m_x that captures only the information that can actually be queried using map and no other information (see Figure 3). That is, m_x is a minimal schema that still allows us to obtain all query results obtainable by running map against m. In addition to the new schema m_x, we need a database transformation m_m_x that tells us how the data of m can be migrated to m_x. After migrating all instances of m to m_x, we can reformulate our original query map to run against m_x, by composing the reverse transformation 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 4 (EXTRACT) *Let map be a mapping from m. $\langle m_x, m\_m_x\rangle$ = Extract(m, map) holds iff*

i. $m\_m_x \circ$ *Invert*$(m\_m_x) \circ map = map$

ii. $m_x =$ *Range*(m_m_x)

iii. *m_x is a minimal model satisfying (i) and (ii).* □

To tie the definition to Example 8, observe that m_m_x is the database transformation from m to the new schema m_x, while 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. Conditions (ii) and (iii) ensure that m_x does not capture irrelevant information.

Our definition of Extract builds on the (materialized) view selection problem [2, 13, 20, 34], 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). The works 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, 15]. That is, given a view m_m_x, the goal is to rewrite query map on m into query $q =$ Invert$(m_m_x) \circ map$ on m_x.

Definition 4 covers a general case in which map is an arbitrary, possibly non-functional mapping. That is, map may express arbitrary constraints between m and some model m'. If map is a query on m, Extract returns a minimal model that can hold the results of the query. This case is illustrated below.

EXAMPLE 9 Let

$m = \ll$R($\underline{\text{ID}}$: int, Age : int)$\gg$
$map = \ll$SELECT Age FROM R
WHERE Age=18 OR Age=19$\gg$

Then, the result of Extract can be specified as

$m_x = \ll$S(A : bool)$\gg$
$m_m_x = \ll$SELECT 19−Age FROM R
WHERE Age=18 OR Age=19$\gg$ □

Observe that condition (i) of Definition 4 is satisfied trivially when m_m_x is total and injective. In this case, composing m_m_x with its inverse yields the identity. The intuition is that it is always possible to find a view schema that supports the given query workload by simply picking the original schema $m_x = m$ and bijection $m\_m_x = \mathsf{Id}(m)$. The minimality condition (iii) guards against such trivial solutions.

THEOREM 2 Extract has the following properties:

2A. The output model m_x and mapping m_m_x of Extract are determined up to isomorphism. If

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

then the bijection from m_x onto n_x is

$m_x\_n_x = \mathsf{Invert}(m\_m_x) \circ m\_n_x$

2B. If $\langle m_x, m\_m_x\rangle = \mathsf{Extract}(m, map)$ then $|m_x| = |\mathcal{P}(map)|$, where $\mathcal{P}(map)$ is a partitioning of $\mathsf{Domain}(map)$ by the equivalence relation $ind(x_1, x_2) =_{\text{df}} (map[x_1] = map[x_2])$.

2C. If m_m_x is a surjective view from m onto m_x, then there is no way to "compress" the view schema m_x any further, i.e.,

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

2D. If all of m participates in map, then the extracted model is isomorphic to the input model. Formally, if $\mathsf{Invert}(map)$ is a surjective function and $\langle m_x, m\_m_x\rangle = \mathsf{Extract}(m, map)$, then m_m_x is a bijection. □

PROOF SKETCH: The proof of the theorem follows the same steps as that of Theorem 1. First, we provide an alternative characterization of Extract (see Lemma 2 below). Then, we prove the theorem by constructing and analyzing instance graphs for Extract using the lemma.

LEMMA 2 Let $\mathsf{Domain}(map) \subseteq m$. $\langle m_x, m\_m_x\rangle = \mathsf{Extract}(m, map)$ holds iff:

1. m_m_x is a surjective function from m onto m_x
2. For all $(y_1, x_1), (y_2, x_2) \in m\_m_x$: $x_1 = x_2$ iff $map[y_1] = map[y_2]$
3. $\mathsf{Domain}(m\_m_x) = \mathsf{Domain}(map)$ □

The lemma replaces condition (iii) of Definition 4 by condition (2), which can be tested locally. It shows that the essence of Extract is to collapse the states of m that are "indistinguishable" under map into a single state of m_x.

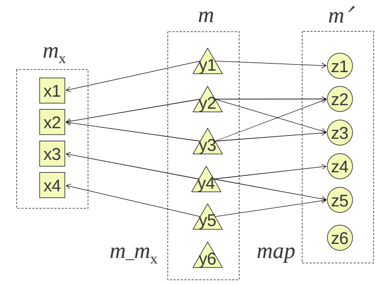

Figure 4: Instance graph illustrating Extract

The partitions of $\mathcal{P}(map)$ in Part 2B hold such indistinguishable states, so that m_x contains one state per partition. Condition (3) makes sure that exactly those instances of m that are connected in map are those that participate in m_m_x. □

EXAMPLE 10 Figure 4 shows an instance graph that illustrates applying the operator Extract to a model m with six instances. Instances y_2 and y_3 are indistinguishable under map since $map[y_2] = map[y_3] = \{z_2, z_3\}$. Therefore, they are collapsed into a single instance x_2 of m_x. All other instances of m are pairwise distinguishable. Instance y_6 is not connected in map and thus has no counterpart in m_x. Observe that the output mapping m_m_x differs structurally from the input mapping map, i.e., Extract specifies a non-trivial mapping transformation. □

Part 2D of Theorem 2 yields the following property:

COROLLARY 2 (IDEMPOTENCE) Extraction from an extracted model yields the same model (up to isomorphism). Formally, if $\langle m_x, m\_m_x\rangle = \mathsf{Extract}(m, map)$, then $\langle m_x, \mathsf{Id}(m_x)\rangle = \mathsf{Extract}(m_x, \mathsf{Invert}(m\_m_x))$. □

Algorithms for query reachability (e.g., [16, 33]) 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].

3.4 Diff operator

The operator Diff is complementary to Extract. It takes a model m and a mapping map between m and some model m', and returns a portion m_d of m that does *not* participate in the mapping. Intuitively, "difference" is a minimal model that when merged with an extracted model, produces the original model. We continue with the scenario of Example 8.

EXAMPLE 11 The legacy database with schema m from Example 8 has been migrated to a new operational database with schema m_x that captures only the information that can be queried using map (see Figure 5). Assume that for legal reasons all data in the legacy database has to be

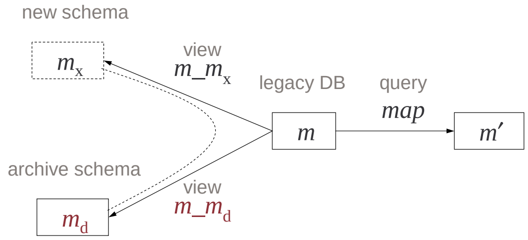

Figure 5: Illustration of Diff (Example 11)

preserved indefinitely. For efficiency, the legacy data is split between the new operational database and an archival database. Our goal is to develop an archival schema m_d that captures only the information needed to reconstruct the legacy data from the new operational database and the archive, and no other information. In addition, we need a database transformation m_m_d to populate m_d with data from m. Together, m_m_d and m_m_x describe how the data in the new operational database relates to the data in the archive, namely that $m_x\_m_d = \mathsf{Invert}(m\_m_x) \circ m\_m_d$. The legacy 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 5 (DIFF) *Let map be a mapping from m. $\langle m_d, m\_m_d \rangle = \mathsf{Diff}(m, map)$ holds iff the following conditions are satisfied for some m_x, m_m_x:*

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

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

iii. *m_d is a minimal model satisfying (i) and (ii)* □

The following example illustrates the effect of the operator for concrete model and mapping definitions:

EXAMPLE 12 Let

m = $\ll \mathrm{R(A, B), S(B, C)};\ \pi_\mathrm{B}(\mathrm{R}) \subseteq \pi_\mathrm{B}(\mathrm{S}) \gg$
map = $\ll \mathrm{T = R \bowtie S} \gg$

Then, the result of $\mathsf{Diff}(m, map)$ can be expressed as

m_d = $\ll \mathrm{V(B, C)} \gg$
m_m_d = $\ll \mathrm{V = S} - \pi_\mathrm{B,C}(\mathrm{R \bowtie S}) \gg$ □

Our definition of Diff is based on the view complement problem [4, 18]. 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.

Definition 5 covers a general case when map is an arbitrary, possibly non-functional or partial mapping. If the input mapping is a total view, the output of Diff corresponds to a (minimal) view complement:

COROLLARY 3 (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, i.e., the following holds:

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

The corollary can be shown by substituting m_m_x for map in Definition 5 and using the result of Theorem 2 (Part 2C). In Example 7, the views m_m_1 and m_m_2 are complementary yet neither is minimal (i.e., does not satisfy Diff), as demonstrated in [24].

THEOREM 3 Diff has the following properties:

3A. The output model m_d of Diff is determined up to isomorphism, whereas the mapping m_m_d is not.

3B. If $\langle m_d, m\_m_d \rangle = \mathsf{Diff}(m, map)$ then $|m_d| = \max\{|C| : C \in \mathcal{P}(map) \cup \{\emptyset\}, |C| \neq 1\} + |m - \mathsf{Domain}(map)|$, where $\mathcal{P}(map)$ is as in Theorem 2.

3C. If all of m participates in map, the "difference" m_d is empty. Formally: if $\mathsf{Invert}(map)$ is a surjective function and $\langle m_d, m\_m_d \rangle = \mathsf{Diff}(m, map)$, then $m_d = \emptyset$ and $m\_m_d = \emptyset$. □

Part 3A tells us that there may be multiple ways of compensating the information loss that occurs upon extraction, a result consistent with [4]. Part 3B states that the output model of Diff contains as many instances as in the largest partition of $\mathcal{P}(map)$, unless the size of the largest partition is 1. In this case, $\mathsf{Invert}(map)$ is a surjective function, i.e., all of m participates in map, so that the size of m_d is zero (Part 3C).

PROOF SKETCH: The proof of Theorem 3 is based on the following lemma, which provides an alternative characterization of Diff that removes condition (iii) of Definition 5. The lemma emphasizes that the essence of Diff is to ensure that the states of m that are indistinguishable under map (and, hence, would be collapsed in Extract) can be distinguished by way of m_m_d.

LEMMA 3 Let $\mathsf{Domain}(map) \subseteq m$. $\langle m_d, m\_m_d \rangle = \mathsf{Diff}(m, map)$ holds iff:

1. m_m_d is a surjective function from m onto m_d
2. For all $y_1, y_2 \in m$ with $map[y_1] = map[y_2]$ and $y_1 \neq y_2$ there exist $(y_1, d_1), (y_2, d_2) \in m\_m_d$ with $d_1 \neq d_2$
3. For all $y \in m - \mathsf{Domain}(map)$ there is $(y, d) \in m\_m_d$ with $\{y' \mid (y', d) \in m\_m_d\} = \{y\}$
4. For each $d \in m_d$ there exists $(y, d) \in m\_m_d$ such that $map[y] = \emptyset$ or $map[y] = map[y']$ for some $y' \neq y$
5. There exists $y' \in m$ such that for each $d \in m_d$ there exists $(y, d) \in m\_m_d$ with $map[y] = map[y']$ or $map[y] = \emptyset$ □

Condition (2) makes sure that the instances of m that are indistinguishable in map become distinguishable in m_m_d. Condition (3) requires that the instances of m that do not participate in map have unique images in m_d. Conditions (4) and (5) ensure that m_d does not contain any irrelevant information. The proof is completed by analyzing the instance graphs constructed using the lemma. □

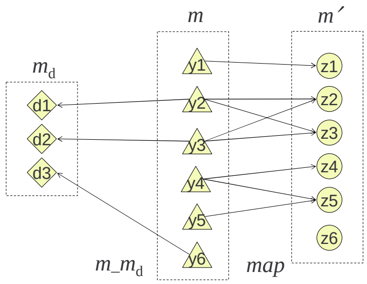

Figure 6: Instance graph illustrating Diff

EXAMPLE 13 Figure 6 shows an instance graph illustrating Diff applied to the model m and mapping map of Example 10 (compare Figure 4). Instances y_2 and y_3 are indistinguishable under map and are therefore mapped to two unique instances d_1, d_2 of m_d. Instance y_6 does not participate in map and is "pulled out" into m_d to avoid information loss. □

The polynomial algorithm of [18] 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.

3.5 Confluence operator

Confluence is a new operator that we developed by analyzing the properties of several model-management scenarios, such as change propagation [6, 25]. 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. Confluence can be thought of as an operator that merges two mappings, as opposed to merging models. It is defined as follows:

DEFINITION 6 (CONFLUENCE, $\oplus$)

$$\begin{aligned} map_1 \oplus map_2 =_{\text{df}} & (map_1 \cap map_2) \\ \bigcup \{(x,y) \in map_1 \mid\ & x \notin \textit{Domain}(map_2) \wedge \\ & y \notin \textit{Range}(map_2)\} \\ \bigcup \{(x,y) \in map_2 \mid\ & x \notin \textit{Domain}(map_1) \wedge \\ & y \notin \textit{Range}(map_1)\} \end{aligned}$$

The operator extracts the "submapping" on which map_1 and map_2 agree and adds to it the correspondences between all those instances of m_1 and m_2 that participate either only in map_1 or only in map_2.

EXAMPLE 14 Let

$m_1 = \ll \mathrm{R}(\mathrm{A},\mathrm{B}),\ \mathrm{S}(\mathrm{B},\mathrm{C}) \gg$
$m_2 = \ll \mathrm{T}(\mathrm{A},\mathrm{B},\mathrm{C}) \gg$
$map_1 = \ll \mathrm{R} \bowtie \mathrm{S} = \mathrm{T} \gg$
$map_2 = \ll \pi_{\mathrm{A}}(\mathrm{R}) = \pi_{\mathrm{A}}(\sigma_{\mathrm{C}>5}(\mathrm{T})) \gg$.

The confluence $map_1 \oplus map_2$ can be expressed as

$\ll \mathrm{R} \bowtie \sigma_{\mathrm{C}>5}(\mathrm{S}) = \mathrm{T} \gg$ □

In the example, $\mathsf{Range}(map_1) \subset \mathsf{Range}(map_2) = m_2$. In this case, $map_1 \oplus map_2 = map_1 \cap map_2$, i.e., the result can be expressed as a conjunction of constraints in map_1 and map_2. This and other properties of Confluence are summarized in Theorem 4.

THEOREM 4 Confluence has the following properties:

4A. $map_1 \oplus map_2 = (map_1 \cap map_2) \cup \big((map_1 \cup map_2) - \mathsf{Domain}(map_1) \times \mathsf{Range}(map_2) - \mathsf{Domain}(map_2) \times \mathsf{Range}(map_1)\big)$

4B. Confluence is commutative, i.e., $map_1 \oplus map_2 = map_2 \oplus map_1$, but not associative.

4C. If the domain and range of one mapping is contained in the respective domain and range of another mapping, then $map_1 \oplus map_2 = map_1 \cap map_2$.

4D. If domains and ranges of mappings are disjoint, then $map_1 \oplus map_2 = map_1 \cup map_2$.

4E. If $\mathsf{Invert}(map_1)$ is injective or domains of map_2 and map_3 are disjoint, then the distributive law holds, i.e., $map_1 \circ (map_2 \oplus map_3) = (map_1 \circ map_2) \oplus (map_1 \circ map_3)$.

4F. The bijection between isomorphic models produced by Merge (Theorem 1, Part 1A) can be expressed using Confluence. Formally, if

$\langle m, m\_m_1, m\_m_2 \rangle = \mathsf{Merge}(m_1, m_2, m_1\_m_2)$;
$\langle n, n\_m_1, n\_m_2 \rangle = \mathsf{Merge}(m_1, m_2, m_1\_m_2)$;

the bijection between m and n can be specified as

$$\begin{aligned} m\_n = & (m\_m_1 \circ \mathsf{Invert}(n\_m_1)) \oplus \\ & (m\_m_2 \circ \mathsf{Invert}(n\_m_2)); \quad \square \end{aligned}$$

4 Specifying the semantics of Rondo

The main value of state-based semantics is to guide the design and analysis of model-management operators for particular schema and mapping languages. This helps us build a model-management system. It also helps us specify its semantics to users, so they can understand the effect of mappings they generate via scripts.

To illustrate the utility of state-based semantics for this kind of design and analysis, we use it to characterize the behavior of our prototype model-management system Rondo. The Rondo paper [25] precisely specifies the metadata artifacts produced as output by the operators, but it does not specify a state-based semantics for Rondo's mapping language, called morphisms. Here, we define the state-based semantics for a subset of that language, called path-morphisms, and argue that Rondo works correctly on them.

Path-morphisms We start with some preliminary definitions: A *morphism* is a set of arcs (called inter-schema correspondences in [27]) connecting the *elements* of two schemas, such as XML types or relational attributes. A *relational tree schema* is a schema in which (i) each relation has a primary key (PK), (ii) for each relation R, at most one relation S has a foreign key (FK) for R, and (iii) for each PK-FK relationship the following constraint also holds: every primary key value is referred to by a foreign key. A tree schema comprises a forest of trees whose nodes are relations and arcs are PK-FK relationships. Essentially, each

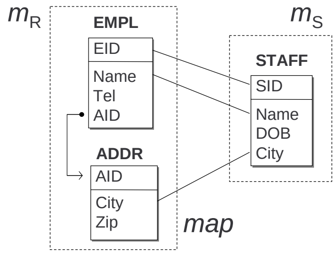

Figure 7: A morphism between two relational schemas

tree in a tree schema is a nested relation, or a snowflake schema as used in data warehousing.

Trees r, s of tree schemas m_R, m_S are *connected* by morphism map if map contains an arc between an element of r and an element of s. Given trees r and s connected by map, the *join key* $JK(r, s)$ is defined as the key R.ID of some relation R in r such that R.ID is connected to a key in s and all attributes of r that are connected to s belong to R or its descendants. The join key $JK(r, s)$ is determined uniquely, if it exists.

Now we define path-morphisms and an interpretation function $I(map)$ for them. $I(map)$ provides a relational algebra expression specifying the state-based semantics of path-morphism map.

DEFINITION 7 (PATH-MORPHISM) *Let map be a morphism connecting tree schemas m_R and m_S. If map connects each tree of one schema to at most one tree of the other schema and for each pair of connected trees r, s there exist join keys $JK(r, s)$ and $JK(s, r)$, then map is called a* path-morphism. *If map is a path-morphism, then for each arc l connecting an attribute* R.A *in r with* S.B *in s, $expr(l)$ denotes the constraint $\pi_{\mathrm{JK(r,s),R.A}}(\mathrm{path_r}) = \pi_{\mathrm{JK(s,r),S.B}}(\mathrm{path_s})$, where $path_r$ ($path_s$) is the join path from R (S) to the table that contains $JK(r, s)$ ($JK(s, r)$). $I(map)$ is the conjunction of constraints $expr(l)$ over all arcs l of map.*

The above definition is really more than a definition: It gives a simple algorithm for testing whether a morphism is a path-morphism and for generating the relational algebra expression $I(map)$ whenever map is a path-morphism.

EXAMPLE 15 Consider relational schemas m_R and m_S and morphism map in Figure 7. It is easy to verify that both schemas are tree schemas and map is a path-morphism such that $I(map)$ is the conjunction of the following individual constraints:[2]

1. $\pi_{\mathrm{EID}}(\mathrm{EMPL}) = \pi_{\mathrm{SID}}(\mathrm{STAFF})$
2. $\pi_{\mathrm{EID,Name}}(\mathrm{EMPL}) = \pi_{\mathrm{SID,Name}}(\mathrm{STAFF})$
3. $\pi_{\mathrm{EID,City}}(\mathrm{EMPL} \bowtie \mathrm{ADDR}) = \pi_{\mathrm{SID,City}}(\mathrm{STAFF})$ □

Let $\mathcal{L}_{\mathcal{R}}$ be the language whose schemas are tree schemas and mappings are path-morphisms. Although $\mathcal{L}_{\mathcal{R}}$ 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.

Sound answers To show that Rondo works correctly for schemas and mappings in $\mathcal{L}_{\mathcal{R}}$, we would like to show that the Rondo operators satisfy the definitions of Section 3 for $\mathcal{L}_{\mathcal{R}}$. However, this turns out not to hold, because Rondo operators do not produce minimal output. Therefore, we define a weaker notion of correctness called *sound answers*.

DEFINITION 8 (SOUNDNESS) *A* sound variable substitution *for a model-management script is a substitution that makes the script a true formula when the operators Extract, Diff, and Merge are replaced by equivalent operators whose definitions do not contain minimality conditions. A* sound answer *to a model-management script are the values of the output variables in a sound substitution. An implementation of model management is* sound *if it produces a sound answer to every script.*

The essence of sound answers is to allow a model-management system to produce non-minimal models. Sound answers are especially useful when the exact answers are too hard to compute, are not representable in the chosen language, or have a very complex representation. To justify sound answers, consider the following literature: [20] selects views that are minimal relatively to a given set of views, but not w.r.t. all conceivable views (i.e., non-minimal Extract is acceptable); [18] 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. Thus, we adopt sound answers as the correctness criterion for Rondo. Clearly, each (exact) answer to a model-management script is sound. The reverse is not true.

EXAMPLE 16 Let

m = $\ll\mathrm{R}(\underline{\mathrm{ID}}, \mathrm{A}, \mathrm{B})\gg$
map = $\ll$SELECT ID,A FROM R$\gg$

Then the following variable substitution produces a sound (but not exact) answer for $\langle m_d, m\_m_d\rangle = \mathsf{Diff}(m, map)$:

m_d = $\ll\mathrm{S}(\underline{\mathrm{ID}}, \mathrm{B})\gg$
m_m_d = $\ll\mathrm{S} = \pi_{\mathrm{ID,B}}(\mathrm{R})\gg$

This sound answer guarantees that we can reconstruct all data stored in R(<u>ID</u>, A, B) from the result of the query map and the data loaded into m_d by way of m_m_d. However, as shown in [18], m_d is not minimal and m_m_d is not a minimal view complement to map. □

Although we only require sound answers, an implementation should still strive for minimality. Moreover, the minimality conditions are critical for expressing the intended semantics of the operators and making them non-redundant. For example, eliminating condition (iii) from Definition 5 would make Diff a derived operator specified in terms of Extract, Merge, Invert, and Compose.

[2] In this example, constraint (1) is entailed by (2) and is redundant.

Rondo is sound The following proposition states the main result of this section. Since Match has no formal semantics, we assume that Match is applied to obtain all morphisms required as input prior to executing the script:

PROPOSITION 2 If the morphisms that are inputs to a script are path-morphisms and are closed under Compose, Confluence, and Invert, then Rondo is a sound implementation of model management.

The closure criterion is based on Definition 1 and 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 result is a path-morphism using the algorithm implied by Definition 7. This criterion is needed because $\mathcal{L}_{\mathcal{R}}$ is not closed under Compose and Confluence. To illustrate, suppose that schema m_R of Figure 7 is connected by the obvious 1:1 morphism map' to a schema m'_R which is identical to m_R but lacks the PK/FK constraint on AID. Then, $map' \circ map$ cannot be expressed as a path-morphism.

PROOF SKETCH: The proof of Proposition 2 involves the following steps. First, we can show that if the result of composition and confluence are in $\mathcal{L}_{\mathcal{R}}$, then the respective Rondo operators produce a path-morphism that represents an exact (and, hence, sound) answer. Second, we show that the output morphisms of all operators are path-morphisms, and that these output morphisms are closed under composition and confluence with the previously computed path-morphisms and those given as input. Finally, if the inputs to Extract, Diff, Merge, and Invert are in $\mathcal{L}_{\mathcal{R}}$, then the result of each operator is sound and is in $\mathcal{L}_{\mathcal{R}}$.

To illustrate the line of argument for the individual operators, consider Extract. Speaking informally, Rondo's extraction algorithm pulls out all attributes referenced in the input morphism together with the relevant constraints. Thus, the output schema is sound since the constraint expression $I(map)$ for the input morphism references only the connected attributes. Since Rondo preserves the constraints in the output schema, the output mapping is guaranteed to be expressible as a path-morphism. Operator Diff produces a sound answer because the primary keys of all connected tables are preserved in the result allowing us to reconstruct the original instances from the results of Extract and Diff. Rondo's conflict-resolution strategy in the Merge algorithm is driven using the direction flags placed on morphism arcs and can potentially result in loss of expressive power upon merging. To eliminate this effect, the direction flags on the input morphism can be adjusted prior to running Merge such that the result contains the least-constrained structures, as suggested in [32]. Invert is exact since Definition 7 is symmetric in the input schemas. □

In many cases, a sound answer produced by Rondo is an exact answer if we assume that the key values in the tables do not bear information, i.e., can be replaced by a permutation of values without affecting the meaning of the data. This assumption is typically valid for auto-generated keys. For the example of Figure 7, if ADDR.AID is an auto-generated key, then Extract(m_R, map) and Extract$(m_S,$ Invert$(map))$ each yield an exact result.

Given that the closure property of the input path-morphisms can be tested algorithmically, Rondo can guarantee that the answers computed by script execution and delivered to a human engineer are sound. Since the state-based semantics of the output path-morphisms is well-defined, they can be deployed in metadata applications to do data migration (if morphisms are functional) or constraint checking.

5 Conclusions and outlook

We presented a state-based approach to studying the semantics of model-management operators, which lays a foundation for a formal treatment of many model-management problems and is instrumental for building future model-management systems. We used the approach to precisely specify the state-based semantics of the Rondo prototype, which we believe is the first such specification of any model-management interface.

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. One can then apply the abstract characterization to concrete languages, as we did for Rondo's language, tree-schemas and path-morphisms. We would like to develop more powerful model-management systems than Rondo, based on richer languages, such as SQL views or GLAV mappings. The abstract state-based characterizations will help us design model-management operators for such languages by giving the technical requirements that those operators must satisfy.

Script optimization requires a deep understanding of the algebraic properties of operators. We presented an initial study in Section 3, but we believe that the full potential for script rewriting is yet to be identified.

Completeness of the suggested set of operators is an interesting open design issue. Analyzing it could help us answer two longstanding questions: (a) what problems can or cannot be solved using model-management operators and (b) are the suggested operators "best"? Under state-based semantics, operators are applied to sets (models) and relations (mappings). Hence, completeness of operators could be studied in a way similar to relational completeness, but on a different meta-level. The notion of completeness has to take into account that the output of Merge, Extract, and Diff is not determined uniquely.

An intriguing extension of our formalization is obtained by considering n-ary mappings, such as $map \subseteq m_1 \times m_2 \times \ldots \times m_n$. As noted in [22], the relationship between two models cannot always be described using a single binary mapping, in which case a "helper" model needs to be used. An example of a helper model is an upper-level ontology that relates two domain-specific ontologies. A mapping established by way of helper models can be viewed as an n-

ary mapping, and suggests further study of the expressive power of n-ary vs. binary mappings. The work [23] also indicates that n-ary mappings may have a greater expressive power. One concrete language for specifying n-ary mappings is AIG [14], where a mapping is a multi-source SQL query yielding an XML document.

References

[1] S. Abiteboul, C. Beeri. The Power of Languages for the Manipulation of Complex Values. *VLDB Journal*, 4(4), 1995.

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

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

[4] F. Bancilhon, N. Spyratos. Update Semantics of Relational Views. *TODS*, 6(4), 1981.

[5] BEA, IBM, Microsoft. Business Process Execution Language for Web Services Version 1.0, 2002.

[6] P. A. Bernstein. Applying Model Management to Classical Metadata Problems. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*, 2003.

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

[8] P. A. Bernstein, E. Rahm. Data Warehouse Scenarios for Model Management. In *Proc. ER*, 2000.

[9] J. Biskup, B. Convent. A Formal View Integration Method. In *Proc. SIGMOD*, 1986.

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

[11] D. Calvanese, G. D. Giacomo, M. Lenzerini, M. Y. Vardi. Lossless Regular Views. In *Proc. PODS*, 2002.

[12] M. A. Casanova, V. M. P. Vidal. Towards a Sound View Integration Methodology. In *Proc. PODS*, pages 36–47, 1983.

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

[14] W. Fan, M. Benedikt, C.-Y. Chan, J. Freire, R. Rastogi. Capturing both Types and Constraints in Data Integration. In *Proc. SIGMOD*, 2003.

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

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

[17] R. Hull. Relative Information Capacity of Simple Relational Database Schemata. *SIAM Journal on Computing*, 15(3), 1986.

[18] J. Lechtenbörger, G. Vossen. On the Computation of Relational View Complements. *TODS*, 28(2), 2003.

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

[20] C. Li, M. Bawa, J. D. Ullman. Minimizing View Sets without Loosing Query-Answering Power. In *Proc. ICDT*, 2001.

[21] J. Lin, A. O. Mendelzon. Merging Databases Under Constraints. *Intl. Journal of Cooperative Information Systems*, 7(1), 1998.

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

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

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

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

[26] R. J. Miller, Y. E. Ioannidis, R. Ramakrishnan. Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. *Information Systems*, 19(1), 1994.

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

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

[29] E. Rahm, P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4), 2001.

[30] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, J. Funderburk. Querying XML Views of Relational Data. In *Proc. VLDB*, 2001.

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

[32] S. Spaccapietra, C. Parent. View Integration: A Step Forward in Solving Structural Conflicts. *Trans. on Knowledge and Data Eng. (TKDE)*, 6(2), 1994.

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

[34] D. Theodoratos, S. Ligoudistianos, T. K. Sellis. View Selection for Designing the Global Data Warehouse. *Data and Knowledge Eng. (DKE)*, 39(3), 2001.