Model Management 2.0: Manipulating Richer Mappings

Philip A. Bernstein, Sergey Melnik
Microsoft Corporation
One Microsoft Way, Redmond, WA 98052-6399 U.S.A.
{philbe, melnik}@microsoft.com

ABSTRACT
Model management is a generic approach to solving problems of data programmability where precisely engineered mappings are required. Applications include data warehousing, e-commerce, object-to-relational wrappers, enterprise information integration, database portals, and report generators. The goal is to develop a model management engine that can support tools for all of these applications. The engine supports operations to match schemas, compose mappings, diff schemas, merge schemas, translate schemas into different data models, and generate data transformations from mappings.

Much has been learned about model management since it was proposed seven years ago. This leads us to a revised vision that differs from the original in two main respects: the operations must handle more expressive mappings, and the runtime that executes mappings should be added as an important model management component. We review what has been learned from recent experience, explain the revised model management vision based on that experience, and identify the research problems that the revised vision opens up.

Categories and Subject Descriptors
H.2.5 [**Heterogeneous Databases**]

General Terms
Algorithms, Design, Theory

Keywords
data exchange, data integration, data translation, model management, schema evolution, schema matching, schema mapping, engineered mapping

1. INTRODUCTION
One of the main goals of database management is to make it easier for users to write programs that access large shared databases. We call this the *data programmability problem*. One reason why data programmability is not easy is that it often requires complex mappings between different representations of data. Those different representations arise for two main reasons: heterogeneity and impedance mismatch. Heterogeneity arises because data sources are independently developed by different people and for different purposes and subsequently need to be integrated. The data sources may use different data models, different schemas, and different value encodings. Impedance mismatches arise because the logical schemas required by applications are different from the physical ones exposed by data sources [31]. In both cases, much of the work to access the data involves designing, implementing, testing, and using mappings between these different data representations. The subject of this paper is how to make this work easier.

SIGMOD'07, June 12–14, 2007, Beijing, China.

Integrating heterogeneous data is among the oldest of database problems. It predates "SIGMOD," which was called SIGFIDET (for FIle DEscription and Translation) before being renamed SIGMOD in 1975. Every database research self-assessment has listed interoperability of heterogeneous data as one of the main problems where more research is needed [2][12][13] [94].

The database field has been quite successful in addressing the data programmability problem. Data integration, the problem of providing access to heterogeneous data sources, has been a popular research topic for 25 years [34][95][100]. There is a huge research literature on solutions to the heterogeneity and impedance mismatch problems. And there are many products to help solve those problems.

However, despite this progress, coping with heterogeneity and impedance mismatch remains one of the most time-consuming data management problems. Anecdotal evidence suggests that it is 40% or more of the work in enterprise IT departments. One study of development projects found that coding and configuring object-to-relational mappings was 30-40% of the effort [58]. It is a large and growing part of scientific, engineering and medical computing. It is needed for many web searches. And it is the essence of the semantic web vision. In short, it is a problem in need of better solutions.

1.1 The Nature of Schema Mappings
To cope with heterogeneity and impedance mismatch, the core problem is in developing and using complex mappings between schemas. The nature of the problem depends a lot on the amount of precision required in the mapping specification.

In enterprise IT and many other domains, one needs to specify an *engineered mapping* between the schemas of the data to be accessed or integrated. By "engineered," we mean that the mapping is precisely specified and tested for each application. We'll use the term *data architect* for the role of the person developing engineered mappings.

At the other end of the spectrum lies *approximate mappings*, where users find relationships between data as they go, as in web search or in mining a heterogeneous set of data sources. In these cases, imprecision is tolerable since there is usually no well-defined notion of correct answer. In some cases, a probabilistic analysis may be able to give a formal estimate of the accuracy of

the mapping. But in the end, it is usually up to the user—often an end-user, not a skilled data architect—to determine if the retrieved data is useful.

In between these two ends of the spectrum are cases where both engineered and approximate mappings are developed. For example, data integration is sometimes performed incrementally, where some mappings are carefully engineered and others are done as a best-effort. This approach, called "dataspaces" in [43] [52], arises in data exploration scenarios, such as the management of scientific data, personal information, and military command and control, and in schema extraction from text [47].

The spectrum from engineered to approximate mappings is quite broad. Many points along that spectrum are described in position papers at a recent workshop on data integration [101]. Although the entire spectrum is of great practical importance, we will focus on just one end of it, that of engineered mappings.

There are many usage scenarios that require engineered mappings. One way to characterize them is to list the types of tools used to support them. The following are some common tools where engineered mappings play a central role[1]:

- *Extract-Transform-Load* (ETL) tools, to simplify the programming of scripts to extract data from sources, clean it, reshape it, and load it into a data warehouse [36][59].
- *Message mapping* tools, to simplify the programming of message translation between different formats. These are often embedded in message-oriented transactional middleware, such as enterprise application integration (EAI) environments [5][9][71][98].
- *Query mediators* to access heterogeneous databases. In database research, this is called *data integration* [63]. In commercial IT, it is called Enterprise Information Integration (EII) [51], where there are many variations, e.g., supporting web services and updates [26]. There are custom implementations for bio-informatics and medical informatics [33][35]. This usage scenario may also be served by keyword search.
- *Wrapper generation* tools, for example, to produce an object-oriented wrapper for a relational database [4][54][79]. Unlike query mediators, wrappers often need to support incremental updates. Some enterprise application products include custom tools for this, since the wrappers are such a large piece of the application.
- *Graphical query design* tools, to define a mapping between source and target schemas [84].
- *Portal design* tools, to map data sources to controls that can be conveniently displayed [9][73][90].
- *Forms managers* to map between structured data sources and forms [56][72]. Many enterprise application products include custom tools for this.
- *Report writers* that map between structured data sources and a report format [32][74].
- *OLAP databases*, which map data sources into data cubes that are suitable for OLAP queries [29].
- *Data translation* tools for moving data between different applications [72]. For commercial applications, this role has been partly subsumed by ETL tools. However, for design tools it is a separate product category. For example, mechanical CAD tools need to translate between different geometric coordinate systems, assembly structures, and data formats [23].

Despite the obvious overlap in mapping functionality between these tools, there is little shared mechanism between them, in some cases even when offered by the same vendor.

1.2 The Problem

Given the existence of all these tools, why is it still so labor-intensive to develop engineered mappings? To some extent, it is an unavoidable consequence of ambiguity in the meaning of the data to be integrated. If there is a specification of the schemas, it often says little about integrity constraints, units of measure, data quality, intended usage, data lineage, etc. Given that the specification of meaning is weak and the mapping must be precisely engineered, it seems hopeless to fully automate the process anytime soon. A human must be in the loop.

Since human designers are required, the solution must lie in raising the level of abstraction in which engineered mappings are specified and in offering better tools to do that specification. We need better tools to help the data architect understand the semantics of the data to be integrated, select data sources, extract schema from unstructured sources, deduplicate overlapping data, clean up inconsistencies, choose among different types of integration tools (ETL, EII, replication), design and implement mappings, debug mappings, expose mapping provenance, and revise mappings when schemas evolve. These problems and others were nicely summarized by Laura Haas in [48].

Most of these problems are hard. A lot of engineering effort is required to build tools to solve them. To maximize the functionality of the tools that can be built with a given engineering budget, we need reusable components that can be applied to a wide variety of scenarios. We already do this for the execution environment, notably with query execution engines, which are usually part of a database system or middleware framework. We need to do this for the design-time environment too—we need to produce reusable components of tools.

1.3 The Initial Research Agenda

One component that is present in all of the tools listed in Section 1.1 is a mapping designer. This component helps the data architect design a mapping between schemas in a high-level notation. It generates code that implements the mapping, typically in a programming language or query language, depending on the scenario. Ideally, it should also help the user evolve a mapping after one of the mapped schemas changes, though this is not commonly offered today.

The need for a more powerful mapping designer was recognized by Miller, Haas, and Hernández [75] in the first of a long series of papers about the Clio project (e.g., [49][76][99][102]). The project has explored ways to simplify the data architect's job by proposing mappings based on simple correspondences between elements of the source and target schema, generating code from the mappings, and updating the mappings when one of the schemas changes [103]. The tool can be used to generate executable mappings in a range of languages, such as SQL, XQuery, or XSLT. Some of the technology is now available in IBM Rational Data Architect [57][89].

An alternative to building a general-purpose mapping designer is to build an engine for schema and mapping manipulation func-

[1] Of the large number of products in each category, we cite a somewhat random and small subset that happen to be known to us. We apologize for all of the omissions.

tions that are common to a wide variety of tools for data programmability. In [16][17] we proposed such an engine, called a model management system, and refined the proposal in [10]. Model management supports operations to match schemas, merge schemas, translate schemas, diff schemas, and compose mappings. It is generic in the sense that it supports multiple metamodels and mapping languages.

1.4 The Revised Research Agenda

Initially, this model management approach seemed rather different than the mapping designer approach of Clio. However, over time, the two approaches have converged and are exploring essentially the same problem space. Let us see how this came about.

The original model management proposal was influenced by the first author's experience with Microsoft Repository [11]. That system was meant to support tools for application and database design and development. The tools were meant to use Microsoft Repository for impact analysis, dependency management, configuration management, static lineage, and other functions that required only simple relationships between artifacts.

Since the manipulation of simple relationships is well understood, the initial model management proposal used a mapping language based on them, rather than a highly expressive mapping language that would require solutions to difficult mathematical problems, such as composing and merging mappings expressed in a predicate calculus language. The simple mapping language was designed to be easy to manipulate, factoring out the problem of manipulating complex expressions that have instance-level semantics. Indeed, the first implementation of a model management system followed this approach [69]. We hoped that implementations could eventually offer extensibility hooks for plugging in and manipulating more expressive languages.

So far, this hope has not been realized by most of our experience in applying model management to practical problems. Instead, we have usually found it easier to build custom implementations for expressive mapping languages and solve the mathematical problems that this implies. In part, this is due to the choice of practical problems we have tackled—problems of data integration and wrapper generation, not of design and development tools—which require expressive mappings. In part, it is also due to the difficulty of developing a generic expressive mapping language and applying it to different metamodels. Some would argue that this result is inevitable; a variety of approaches to generating and manipulating engineered mappings is necessary due to the wide range of data programmability problems being addressed. While it may turn out this way, we still have reason to believe that a generic model management engine is feasible. But it requires developing model management operators that manipulate highly expressive mappings, which was not the original vision. This is one reason why our vision for model management has changed.

Another outcome of our experience in applying model management to practical problems is the need for more focus on the runtime system that supports the execution of mappings. The runtime system does not simply execute queries over mappings. It must also propagate updates, notifications, exceptions, and access rights, and provide other services, such as debugging, synchronization, and provenance. These problems are sensitive to the expressiveness of mappings and to the capabilities of the model management operators that generate the mappings. That is, the ability to support a certain amount of expressiveness in mappings depends not only on design-time capabilities of a model management system to manipulate those mappings but also on runtime capabilities to provide services over those mappings. Given these interdependencies, the runtime support for mappings needs to be considered as part of the model management system. This too has caused us to rethink our vision.

Recent published work from the Clio group at IBM and their university collaborators has evolved in a similar direction, but from a different starting point. Their early work focused mostly on the mapping design tool. However, since then they have done seminal work on two of the model management operations, Compose [40] and Inverse [37][41], and on the semantics of query answering [38][39], which is closely related to code generation. A summary of this work appears in [60]. Model management research has landed in the same place: Starting with operations on schemas and simple mappings, it has evolved to focus on highly expressive mappings, like Clio.

Given our experience and that of others, it is time to revisit the model management vision to review what has been learned from that experience, to revise the vision based on that experience, and to identify the research problems that the revised vision opens up. Necessarily, much of this will involve summarizing our own work and that of the Clio group.

The next section introduces the abstractions and capabilities of a model management system. Sections 3-6 explore those capabilities in more detail, describing the main operators of model management and summarizing what is known about them. Section 7 is the conclusion.

2. MODEL MANAGEMENT

A *model management system* is a component that supports the creation, compilation, reuse, evolution, and execution of mappings between schemas represented in a wide range of metamodels. The user-oriented goal is to simplify the development and maintenance of applications that perform data programming. However, a model management system (MMS) is not a user-oriented tool. Rather, it is a reusable component that can be embedded, with relatively modest customization, into user-oriented tools for data warehouse loading, message mapping, query mediation, wrapper generation, report writing, and other data programmability problems.

The main abstractions supported by an MMS are schemas and mappings. Since an MMS should be generic, the choice of languages in which to express schemas and mappings is important.

A *schema* is an expression that defines a set of possible instances, that is, database states. A *metamodel* is a language for expressing schemas. To enable reuse for a wide enough range of scenarios, an MMS must support schemas expressed in all popular metamodels. Today, that means SQL, XML Schema (XSD), Entity-Relationship (ER), and object-oriented (OO) metamodels (e.g., Java, ODMG [28], and .NET), and perhaps Service Modeling Language (SML) [91], Resource Description Framework (RDF) [85] and Web Ontology Language (OWL) [80]. Ideally, a basis set of data type constructs that are common to many metamodels could cover most of their features, with only a few specials that are included for one metamodel only. It is not a trivial undertaking to define such a universal metamodel that is elegant and has precise semantics that can be succinctly specified. However, it is clearly doable with some effort and not what stands in the way of building a powerful MMS.

The harder part is in developing technology for an MMS to support mappings between many popular metamodels. It is unclear how best to go about this. One could develop a language that can express mapping constraints between schemas in the universal metamodel. While it is beneficial to have one mechanism like this, the mapping language might have to be rather complex to handle so many different types. Or one could use multiple languages. For example, to map XML to SQL, one could use SQL as a mapping language to pull shredded data from a SQL database, compose that mapping with a default XML representation of the data, and compose the result with an XQuery mapping to reshape the XML.

A mapping expresses a relationship between the instances of two schemas [66]. We can formally define the *instance-level semantics* of a mapping as follows: If $\mathcal{D}_1$ and $\mathcal{D}_2$, are the sets of possible instances of schemas S_1 and S_2 respectively, then a mapping between S_1 and S_2 defines a subset of $\mathcal{D}_1 \times \mathcal{D}_2$ [66][67]. Usually, a mapping is expressed as a set of *mapping constraints* (sometimes called *inter-schema constraints* [27]), each of which is a formula in some *mapping language*; it defines the subset of $\mathcal{D}_1 \times \mathcal{D}_2$ for which the formula holds.

An MMS must support a rich mapping language so it can be applied to a wide variety of scenarios. Given the tension between the expressiveness of mapping constraints and the tractability of manipulating them, choosing the mapping language is a major design challenge. If tractability were not a consideration, one would want a mapping language that includes first-order logic with aggregation, with set and bag semantics, user-defined functions, regular expressions, rich type constructors (e.g., to construct XML fragments), and even heuristic operations such as deduplication.

A *transformation* is a functional mapping constraint, such as a query or view definition. If mappings are restricted to be transformations, and the MMS needs to do nothing more than compile the transformation into executable code, then a highly expressive mapping language may be tractable. However, as we will see, an MMS may need to allow non-functional mapping constraints which it can translate into transformations. Moreover, an MMS must do more than compile mappings. This translation and additional manipulation operations are tractable only if compromises are accepted, such as constraining the expressiveness of mappings or using algorithms that are slow or that make a best effort to solve an intractable problem.

A closely related challenge is the choice of a common language for defining integrity constraints, that is, constraints on one schema (as opposed to mapping constraints that relate two schemas). It needs to be powerful enough to express integrity constraints supported by popular metamodels. Yet it must be feasible to reason over the integrity constraints across mappings. For example, for a given source and target database that are related by a given mapping, we might need to check that if the source database satisfies the source integrity constraints then the target database also satisfies the target integrity constraints.

The functionality of a model management system is encapsulated in its design-time and runtime operations. The main components are shown in Figure 1. They are presented in Sections 3-6, organized as follows:

- Section 3 discusses the generation of mappings either between two given schemas or between a given schema S and a schema generated from S. The main operations are Match and ModelGen.
- Section 4 discusses the generation of transformations from mapping constraints. The main operation is TransGen.
- Section 5 discusses the runtime functions that are needed to support mappings.
- Section 6 discusses problems that arise from schema evolution. The solutions require several additional operations: Compose, Diff, Extract, Merge, and Inverse.

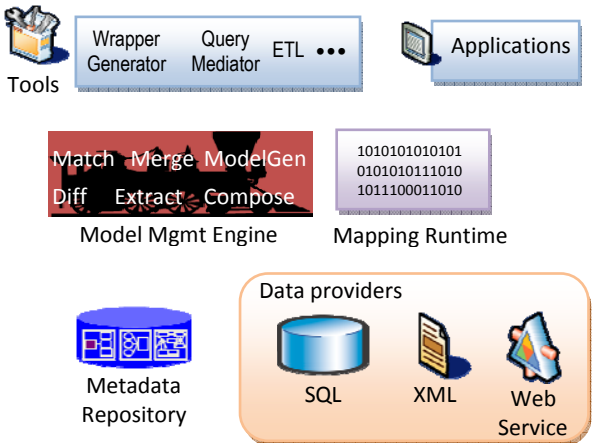


Figure 1: Model Management System Architecture

3. THE ORIGIN OF MAPPINGS

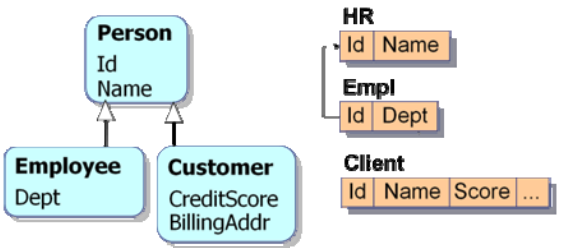
There are two main scenarios for mapping generation, each with variations. In the first scenario, the source and target schemas are given and the data architect defines a mapping between them. For example, the schemas could be a data source and data warehouse schema or message schemas from two business partners. The second scenario is defined by the model management operation called ModelGen; given only one of the two schemas, the other is (semi-) automatically derived along with a mapping between the given schema and the derived schema. For example, the input schema could be a data source schema and the derived schema could be an OO wrapper or form definition. We now discuss each scenario in more detail.

3.1 Given Two Schemas, Generate a Mapping

A good way to think about mapping design is as a three-step process that produces mappings in three successively more refined representations: correspondences, mapping constraints, and transformations. Correspondences are pairs of elements from the two schemas that are believed to be related in some unspecified way. Usually, correspondences do not define a mapping. Rather, they are hints that tell which elements of the two schemas need to be related by a mapping. The second step is to translate those correspondences into mapping constraints [75]. In some cases, the mapping constraints are transformations, so step two completes the process. In other cases, the mapping constraints are not functions, so a third step is required to translate them into transformations.

3.1.1 Schema Matching

The problem of generating correspondences is called *schema matching*. There is a big literature on this topic, offering many different algorithms to compute correspondences [86][92]. They include ways to exploit lexical analysis of element names, schema structure, data types, value distributions, thesauri, ontologies, and previous matches. Most recent work has focused on improving the precision and recall of a schema matcher based on certain types of




```
SELECT p.Id, p.Name                    =   SELECT Id, Name
FROM Persons AS p                          FROM dbo.HR
WHERE p IS OF (ONLY Person)
  OR p IS OF (ONLY Employee)

SELECT e.Id, e.Dept                    =   SELECT Id, Dept
FROM Persons AS e                          FROM dbo.Empl
WHERE e IS OF Employee

SELECT c.Id, c.Name,                   =   SELECT Id, Name,
   c.CreditScore, c.BillingAddr                Score, Addr
FROM Persons AS c                          FROM dbo.Client
WHERE c IS OF Customer
```

Figure 2: Mapping constraints between an ER and SQL schema

schema and instance information. Such work is valuable for approximate data integration, especially in unsupervised settings like the semantic web, and for ontology integration.

However, it is unlikely that improved precision and recall will yield big productivity gains for the data architect who is developing an engineered mapping between independently developed schemas. This is especially true for mapping tasks that are unrelated to previous ones, where there are no validated mappings to reuse. The reason is that much of the data architect's time is spent reading documentation, learning application requirements, writing functions that combine or split element values, and running tests with sample data—activities that are currently beyond the reach of algorithmic solutions. For engineered mappings, we expect that the main value of the matcher is to avoid the need for tedious scrolling around large schemas by offering candidate matches to consider. Thus, a better goal for this setting is to ensure that a matcher returns all viable candidates for a given element, rather than only the best one for every element [18] [46].

The above beliefs are only educated guesses, based on a limited number of discussions we have had with product developers and users. What is missing from the literature are more comprehensive and controlled investigations of how people spend time using a schema matching tool for engineered mappings and, hence, what kinds of features would be most likely to improve their productivity. We believe the biggest productivity gains will come from better user interfaces [42][88], not from more accurate schema matching algorithms. Examples include helping the user focus on the schema elements of interest by dynamically reorganizing them to fit on one screen and providing workflow assistance to track what the user knows about elements that he has already examined.

3.1.2 Mapping Constraint Generation

Given a set of correspondences between two schemas, the data architect needs to generate a transformation, such as a query or view definition over the source schema that populates the target schema.

```
SELECT VALUE -- Constructing Persons
  CASE
    WHEN (T5._from2 AND NOT(T5._from1))
     THEN Person(T5.Person_Id, T5.Person_Name)
    WHEN (T5._from1 AND T5._from2)
     THEN Employee(T5.Person_Id, T5.Person_Name,
                   T5.Employee_Dept)
     ELSE Customer(T5.Person_Id, T5.Person_Name,
          T5.Customer_CreditScore,
          T5.Customer_BillingAddr)
  END
FROM ( (
 SELECT T1.Person_Id, T1.Person_Name,
    T2.Employee_Dept,
    CAST(NULL AS SqlServer.int) AS Customer_CreditScore,
    CAST(NULL AS SqlServer.nvarchar) AS
        Customer_BillingAddr, False AS _from0,
    (T2._from1 AND T2._from1 IS NOT NULL) AS _from1,
    T1._from2
  FROM  (
    SELECT
      T.Id AS Person_Id,
      T.Name AS Person_Name,
      True AS _from2
    FROM dbo.HR AS T) AS T1
    LEFT OUTER JOIN (
    SELECT
      T.Id AS Person_Id,
      T.Dept AS Employee_Dept,
      True AS _from1
    FROM dbo.Empl AS T) AS T2
    ON T1.Person_Id = T2.Person_Id)
  UNION ALL (
  SELECT
    T.Id AS Person_Id,
    T.Name AS Person_Name,
    CAST(NULL AS SqlServer.nvarchar) AS Employee_Dept,
    T.Score AS Customer_CreditScore,
    T.Addr AS Customer_BillingAddr,
    True AS _from0,
    False AS _from1,
    False AS _from2
  FROM dbo.Client AS T)
 ) AS T5
```

Figure 3: A query to populate Persons based on constraints in Figure 2

Some tools automatically generate transformations directly from correspondences. Thus, the correspondences amount to a visual programming language. In some tools the semantics of that language is unclear, so the data architect needs to read the generated transformation to understand the meaning of the correspondences [62].

In our opinion, a better approach is for the mapping design tool to help the data architect translate correspondences into mapping constraints. Each constraint should specify a small enough portion of the desired mapping that the data architect can easily understand what it does and hence determine whether it is what she wants [70].

For example, consider the is-a hierarchy in Figure 2, where Employee and Customer are specializations of Person, which need to be mapped to relational tables HR, Empl, and Client [70]. We can express the mapping constraints as equalities of simple queries, shown in the figure. The queries are expressed in Entity SQL [77], an extension of SQL that can deal with inheritance and other ER concepts. Its syntax uses the keywords IS OF or IS OF ONLY to test whether a variable is of a particular type. The first constraint maps the ID and Name of entities that are either of type Person or Employee to the HR table. The second constraint maps the ID and Dept of entities that are of type Employee to the Empl table. The third maps ID, Name, CreditScore, and BillingAddr of entities of type Customer to the Client table. Each of these constraints is relatively easy to express and understand. However, these constraints imply a rather complex hard-to-understand query on tables that returns data to populate the Person entity set, shown in Figure 3.

This problem of going from correspondences to mapping constraints or queries was explored in IBM's Clio project. In the first Clio paper [75], transformations are generated directly from correspondences. Value correspondences are taken as input, which may include selection predicates and computations over source elements that generate a target element. With some optional user guidance, Clio produces a query. For example, if the source is a relational database schema and the target is a relation schema, then the problem boils down to selecting source relations that have correspondences to the target, choosing joins between the source relations, and possibly adding selections over some of the source relations.

In later papers from the Clio project, mapping constraints are generated from correspondences. For example, in [38] they propose using constraints expressed as source-to-target tuple-generating dependencies, which correspond to global-and-local-as-view (GLAV) formulas [44]. (A more technical definition appears later, in Section 6.1.)

Melnik et al. give a case where correspondences can be unambiguously interpreted as mapping constraints [67]. Intuitively, if the source and target schemas are snowflake schemas as used in data warehousing and the correspondences include one correspondence relating the roots of the two schemas, then each correspondence can be unambiguously interpreted as a mapping constraint that is the equality of two join expressions: one over the source and one over the target. See Figure 4 (taken from [68]). Thus, the data architect only needs to specify correspondences and does not need to translate them into mapping constraints. This simplifies the process of designing mapping constraints, but there's a cost: the set of expressible mappings is quite constrained. It would be useful to find more expressive graphical representations that are relatively simple (like correspondences) and have a precise interpretation as constraints.

Bohannon et al. [24] show how to generate an XML mapping from correspondences that map one DTD to another.

3.2 ModelGen

ModelGen is a model management operation that automatically translates a source schema expressed in one metamodel into an equivalent target schema expressed in a different metamodel, along with mapping constraints between the two schemas.

The first generic (i.e., metamodel-independent) approach we know of is that of Atzeni and Torlone [6]. They introduced the idea of using a repertoire of rules over schemas expressed in a

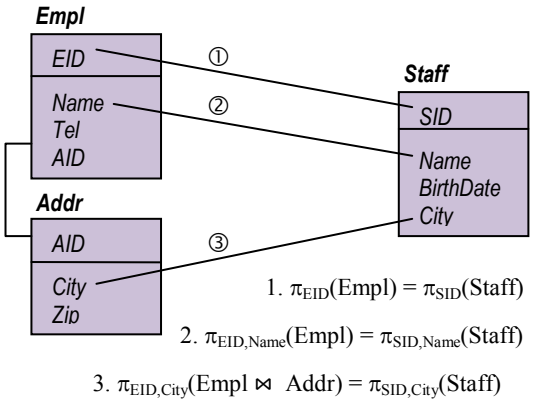


Figure 4: Interpreting Correspondences as Constraints

universal metamodel, where each rule replaces one construct by others. The universal metamodel contains modeling constructs of all metamodels. A sequence of rules is applied to the source schema to eliminate all modeling constructs that are absent from the target metamodel. Their rules are expressed in C++ with abstract signatures that help them determine the correct rule sequence for a given source and target metamodel. They did not generate instance-level mapping constraints.

Two recent projects have extended Atzeni and Torlone's work to, among other things, generate instance translations via three data-copy steps [7][81]: (1) copy the source data into the universal metamodel's format; (2) reshape the data using instance-level rules that mimic the schema transformation rules; and (3) copy the reshaped data into the target system. This approach represents considerable progress, but it has two weaknesses: It is rather inefficient for data exchange. And it still falls short of the need for ModelGen to return declarative mapping constraints between the source and target schema.

An approach to ModelGen that generates declarative mapping constraints is described briefly in [19]. It also describes a flexible mapping of inheritance hierarchies to tables, which is needed for complex enterprise applications. Although there is some claim of genericity in [19], we do not know of a published comprehensive demonstration that mapping constraints can be generated when ModelGen is applied to rich schema languages, e.g., going from SQL to XSD or from XSD to ODMG [28].

McBrien and Poulovassilis describe equivalence-preserving translations of schema constructs in [65][83]. Their goal is data integration rather than schema translation *per se*, but their translation rules may also be applicable to ModelGen.

4. TRANSFORMATION GENERATION

In most of today's tools where engineered mappings play a central role, data architects must design transformations manually, possibly with automated support for generating correspondences using a schema matching algorithm. If we follow the three-step approach described at the beginning of Section 3.1, then data architects would design mapping constraints (as explained in Section 3.1.2), which the mapping design tool translates into executable transformations. We encapsulate this translation activity in an operation called *TransGen*, which produces a transformation that is consistent with the mapping constraints it takes as input.

The type of transformations that are generated depends on the usage scenario. For data exchange, the transformation copies the source database into the target database. For wrapper generation, report writers, and many other scenarios, view definitions are needed to support queries on the target database. For wrapper generation in support of data access applications, the views must also enable updates on the target (i.e., wrapper) schema to be translated into updates on the source.

In some approaches the mapping constraints are not functions from source to target. For example, they may be GLAV constraints. Thus, given a database state that conforms to the source schema, there may be many states of the target database schema that satisfy the constraints. Usually, the desired transformation is a function that creates a target database from a source database. So one of the many target database states that satisfy the constraints must be selected. The approach taken in the Clio project [38][39] is to pick one that has the semantics of certain answers [1]: a query over the target should return only those tuples that are in the output of the query for every target database that satisfies the constraints. In some cases, the desired database state, called a universal instance, contains labeled null values that are needed to compute the answers to queries but are not allowed to be returned as part of the answer.

Another approach is to generate the transformation directly, as in Microsoft's next release of ADO.NET [70]. In ADO.NET, the target is an extended entity-relationship (ER) schema, called the Entity Data Model [22]. The source is a relational database. Users write queries and updates against the target ER schema, which are translated into queries and updates on the relational source database. Each constraint is expressed as an equality condition between two algebraic expressions: one over the target and one over the source, as in Figure 2. These constraints allow inheritance mappings, projections, and selections, but currently do not allow nesting (as in XML) or joins. The paper describes an algorithm that translates the mapping into two view definitions: a query view that expresses the target as a function of the source, which is used to support queries on the target ER schema; and an update view that expresses the source as a function of the target, which is used to translate updates on the ER schema into updates on the relational source database. The views must be lossless. In MMS terms, this says that the composition of the update view with the query view must equal the identity on the target. It is called *roundtripping* since data that passes through the update view and back through the query view is unchanged.

As soon as one moves beyond flat relational mappings, it becomes more difficult to interpret them as transformations. The Clio project has papers explaining how to interpret mappings over XML schemas [45][99], and in [49] how to generate XSLT transformations. In ADO.NET the need for a sophisticated algorithm for generating transformations is in part due to the richness of inheritance mappings.

A lot more work is needed on generating transformations. Constraints need to be enriched to handle more complex mappings. Yet they must still be easy to understand to the data architects who design them. In addition, it must be possible to generate efficient transformations that implement them, which is likely to expose a wealth of optimization opportunities.

5. MAPPING RUNTIME

Most of the literature on problems related to engineered mappings, especially data integration and wrapper generation, assumes that the result of the mapping design is a query or view that relates one or more source schemas S to a target schema $\mathcal{T}$. So the runtime is simply a query processor. However, there are many scenarios that imply other runtime requirements, where actions on data in the context of $\mathcal{T}$ need to be interpreted in the context of S, or vice versa. The difficulty of this interpretation depends on the choice of language for expressing the mapping map_{ST} between S and $\mathcal{T}$. The amount of interpretation that must be done by the runtime depends on how much of it can be done statically by an MMS. For example, consider the following issues:

- *Update propagation* – Allow updates on schema $\mathcal{T}$. These may be expressed in a data manipulation language. Or they may be the result of object-at-a-time updates to cached objects which are later written through to the data sources. In either case, the updates on $\mathcal{T}$ need to be translated into updates on S via map_{ST}.
- *Peer-to-peer* – There is a chain of mappings from the schema to be queried, $\mathcal{T}$, to a source $S1$, which is mapped to a source $S2$, etc. The mapping design tool might optimize a query on $\mathcal{T}$ to collapse the chain into direct mappings, e.g., from $\mathcal{T}$ to $S2$. In any case, the runtime needs to be able to process a query on $\mathcal{T}$ by propagating it through the chain [14][53].
- *Provenance* – After moving data from source to target, a user wants to know the source data that contributed to a particular target data item. This requires design-time analysis of the mapping plus runtime support to assemble a path of data instances that show how the target was derived.
- *Errors* – If a data access via $\mathcal{T}$ is translated into an access on S that generates an error, then the error needs to be passed back through map_{ST} in a form that is understandable in the context of $\mathcal{T}$. For example, in an object-to-relational mapping, an object access may cause an erroneous access to a table that the user of $\mathcal{T}$ doesn't recognize.
- *Debugging* – Like any program, a mapping needs to be debugged. This could be done with breakpoints and single-stepping, which are set in the context of $\mathcal{T}$ but may need to be executed in the context of S. Debugging can also benefit from provenance information that shows how the mapping generated target data (as in [30]), and from intelligent mapping of errors from S to $\mathcal{T}$.
- *Access control* – Access control constraints on the target might be enforced by a combination of constraints enforced on the server and those enforced by the client runtime. This may affect the constraint preprocessing required by the design tools to distribute the access control work between the two layers.
- *Integrity constraints* – Integrity constraints exhibit the same design choices as for access control constraints above. There are both efficiency and feasibility issues when distributing constraint checking between the two layers. That is, due to differences in S's and $\mathcal{T}$'s metamodels, some constraints on $\mathcal{T}$ may not be expressible on S. For example, the disjointness of two sets of instances of two classes in $\mathcal{T}$ with a common superclass is not expressible as relational integrity constraints on S if S is relational and the classes are mapped to distinct tables.
- *Indexing* – It may be desirable to index data that is exposed via $\mathcal{T}$ to support keyword search. However, in a wrapper or query mediator scenario, the data physically resides in the

data sources which have schemas S. For efficiency reasons, it is probably best to index the data sources and derive a mapping that enables the index to be accessed via $\mathcal{T}$.

- *Business logic* – Triggers and other business logic may be attached to data in the context of $\mathcal{T}$. It may be more efficient to execute them in the context of S. This requires pushing the business logic through map_{ST}, which should be done statically.
- *Notifications* – Suppose data is materialized according to $\mathcal{T}$, either fully (e.g., for a data warehouse) or partially (e.g., as a cache). Then it may be valuable for certain actions on data in S to produce notifications of corresponding actions to data in $\mathcal{T}$. For update actions, this is the problem of maintaining materialized views.
- *Synchronization logic* – Data replication rules may be stated in terms of $\mathcal{T}$, e.g., that complex objects in schema $\mathcal{T}_1$ should be replicated to corresponding complex objects in $\mathcal{T}_2$. For efficiency, it may be better to translate the rules into equivalent rules on finer-grained (e.g., relational) data in the corresponding sources S_1 and S_2 to be executed there.
- *Batch loading* – Since most database systems have a high performance interface for batch loading, in many scenarios it would be more efficient to load data directly into S rather than through $\mathcal{T}$. This requires transforming the data to be loaded via map_{ST} into the format required by S 's loader.
- *Data exchange* – Suppose S and $\mathcal{T}$ are logical views of physical schemas $\mathcal{SP}$ and $\mathcal{TP}$, with logical-to-physical mappings $map_{S\text{-}SP}$ and $map_{T\text{-}TP}$. To execute map_{ST} on the physical databases, it may be more efficient to translate it into a transformation $map_{SP\text{-}TP}$ from $\mathcal{SP}$ to $\mathcal{TP}$. If S is a data source and $\mathcal{T}$ is a data warehouse, then the mapping may have interesting characteristics, such as deduplication or other heuristic operators, staging of data in mini-batches, sorting or other blocking operators, and a variety of metamodels such as spreadsheets and pivot tables.

Solutions to many of the above problems are in hand when S and $\mathcal{T}$ are relational schemas and mappings are conjunctive queries. However, there are many open problems when richer data models and mapping languages are permitted.

6. SCHEMA EVOLUTION

When a schema changes, the objects that depend on it may no longer work properly. These dependent objects include views, queries, constraints, and programs that reference the changed schema and databases that are instances of the changed schema.

Many commercial tools to solve the engineered mapping problems of Section 1 require the data architect to develop mappings. As mappings proliferate, the importance of schema evolution is likely to increase as will the need for tools to help.

There are hardly any schema evolution tools today. This is rather surprising since there is a huge literature on schema evolution spanning more than two decades. Why is this? Is it because research solutions are impractical in some way? It would be valuable to have case studies that apply these research solutions to identify their strengths and weaknesses.

Many of the approaches to repairing dependent objects that are affected by schema changes require the manipulation of mappings. These manipulations can be abstracted as sequences of model management operations. We discuss some of these sequences in this section. Along the way, we will cite papers that are directly relevant to the use of model management for schema evolution. A more complete bibliography with references to over 300 schema evolution papers appears in [87].

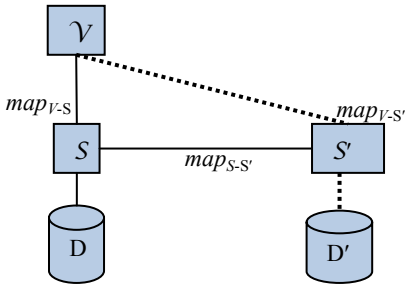


Figure 5: Schema evolution scenario

6.1 Using Composition

Consider the relatively simple schema evolution scenario in Figure 5. Initially, we have schema S that has a database instance D and a view $\mathcal{V}$ defined on S. Now suppose S is modified, yielding S'. What options do we have to cope with that change?

One possibility is to express the change from S to S' as a mapping $map_{S\text{-}S'}$, and to use the mapping $map_{S\text{-}S'}$ first to migrate D to become an instance of S', and then to modify $map_{V\text{-}S}$ so it refers to S' instead of S. How might this be done? The process of developing $map_{S\text{-}S'}$ was discussed in Section 3.1. The process of generating a transformation from $map_{S\text{-}S'}$ to migrate D to become an instance D′ of S' was discussed in Section 4. Thus, the mapping $map_{V\text{-}S'}$ can be obtained by composing $map_{V\text{-}S}$ with $map_{S\text{-}S'}$.

A concrete example is shown in Figure 6. We are given mapping constraint $map_{V\text{-}S}$ between $\mathcal{V}$ and S. Then the Addresses table in S is split into two tables for local addresses and foreign addresses, yielding S'. Since S has changed, mapping $map_{V\text{-}S}$ is no longer valid. To update it, we first represent the change from S to S' as the mapping $map_{S\text{-}S'}$ shown in the figure, and then compose the two mappings $map_{V\text{-}S}$ and $map_{S\text{-}S'}$, yielding the following mapping $map_{V\text{-}S'}$: $\mathsf{Students} = \pi_{\mathsf{Name,Address,\ Country}}\,(\mathsf{Names}' \bowtie (\mathsf{Local} \times \{\text{“US”}\} \cup \mathsf{Foreign}))$

What exactly does the composition operation do? We can express this using *instance-level semantics*, which we introduced in Section 2: each schema S has a set $\mathcal{D}$ of possible database instances and a mapping map_{12} between S_1 and S_2 defines a subset of $\mathcal{D}_1 \times \mathcal{D}_2$. Given mapping constraints map_{12} between S_1 and S_2 and map_{23} between S_2 and S_3, the composition $map_{12} \bullet map_{23}$ is defined to be the set of all pairs of instances $D_1 \in \mathcal{D}_1$ and $D_3 \in \mathcal{D}_3$ such that there exists a $D_2 \in \mathcal{D}_2$ such that $<D_1, D_2>$ satisfies map_{12} and $<D_2, D_3>$ satisfies map_{23} [40][66].

Instance-level semantics precisely defines the behavior of composition in the abstract. However, for the operation to be useful, we need concrete algorithms that can implement it for particular schema and mapping languages. There has been some progress along these lines which we summarize briefly. See also [60].

Suppose a mapping is expressed as a containment of two project-join expressions, $PJ_S \subseteq PJ_T$, over relational schemas, meaning that the set of tuples in the result of the first expression is contained in the result of the second expression. In the theory literature, this is

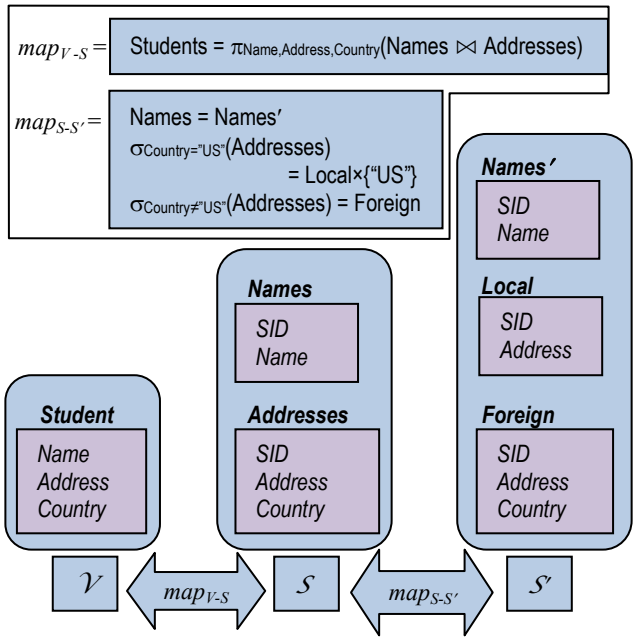


Figure 6: Using mapping composition for schema evolution

called a *tuple generating dependency* [2] (tgd) [3]. If PJ_S refers only to symbols of the source schema and PJ_T refers only to symbols of the target schema, then it is a *source-to-target tgd* (st-tgd) [38].

In [40], Fagin et al. showed that the composition of two st-tgd's is not always expressible as a set of st-tgd's. That is, st-tgd's are not closed under composition. To circumvent this problem, they introduced a second-order extension of st-tgd's that is closed under composition. They give an algorithm to compute that composition, which has an exponential lower bound since the size of the output may be exponential. Their decision to use second-order st-tgd's is another example of how the behavior of design-time model management operations can affect the mapping runtime. That is, they proposed a different mapping constraint language to obtain a well-behaved mapping composition algorithm. If one wants to execute constraints, then the proposal places a requirement either on the mapping runtime to support that language or on design-time operations to translate these constraints into a language that the runtime can execute.

Yu and Popa [103] extend the algorithm of [40] to handle nesting and apply it to some schema evolution scenarios. They also study optimizations of the result of the composition.

In [78], Nash et al. show that the problem of composing mapping expressed as tgd's that may not be source-to-target is undecidable. Nevertheless, they give an algorithm to compute the composition. Clearly, the algorithm does not terminate for all inputs, but when it does it gives the right answer. In [15], Bernstein et al. describe an implementation of that algorithm with some extensions and report on experiments.

From this recent work, we know that mapping composition is a hard problem whose difficulty is quite sensitive to the expressiveness of the allowed mappings. To apply composition in practical settings, we expect that richer languages will need to be considered. Thus, there is much opportunity for further research, both to extend the reach of existing algorithms and to use them to solve practical problems in the real world.

6.2 Diff

Let us revisit the scenario of Figure 5. Suppose S' includes some information that was not expressed by S. What if we want to update $\mathcal{V}$ to include that information? Versions of this scenario are described in [10][17][20][67], with example programs. We summarize one variation briefly here to motivate the need for some of the other model management operations.

First, we need to identify the new parts of S'. This is the responsibility of the Diff operation. It takes S' and $map_{S'\text{-}S}$ as input. Intuitively, it returns a schema S'' that includes the new parts of S' (i.e., the parts of S' that do not participate in the mapping $map_{S\text{-}S'}$) and a mapping $map_{S'\text{-}S''}$ that describes the overlapping parts of S' and S''.

The instance-level semantics of Diff can be described using its dual operation: Extract(S', $map_{S'\text{-}S}$) returns a maximal sub-schema of S' that can be populated with data from S via $map_{S'\text{-}S}$ along with a mapping between that sub-schema and S'. Diff(S', $map_{S'\text{-}S}$) is essentially the complement of Extract. That is, it returns a sub-schema of S' that includes the parts of S' that were not returned by Extract. We omit the instance-level semantics of Diff, which are rather involved; see [67] for details. The first mathematical characterization of Diff that we know of was given in [8] using category theoretic concepts, where it was called the view complement problem. The only algorithm we know of to compute Diff is that of Lechtenbörger and Vossen [61] for the case when the input mapping is a relational select-join view.

Notice that the definition of Diff takes S' and $map_{S'\text{-}S}$ as input, while the mapping given in Figure 5 is $map_{S\text{-}S'}$. This is just a minor syntactic issue that can be fixed by the Invert operation. Recall that $map_{S'\text{-}S}$ defines a subset of $\mathcal{D}' \times \mathcal{D}$, where $\mathcal{D}'$ and $\mathcal{D}$ are the sets of possible instances of S' and S respectively. Invert($map_{S'\text{-}S}$) is defined to be the set of pairs <D, D′> such that <D′, D> is in $map_{S'\text{-}S}$. Thus, to identify the new parts S'' of S', we run Diff(S', Invert($map_{S\text{-}S'}$)).

It is possible that some of the information in S is lost in S'. We can capture this using the concept of information capacity [55]. Roughly speaking, we say that the information capacity of S' is at least that of S if there is a function f on database instances such that for any instance D of S, there is an instance D′ of S' such that f(D′) = D. If the information capacity of S' is not at least that of S, then some information is lost in S'. We can save the data that would be lost in a migration from S to S' by calling Diff(S, $map_{S\text{-}S'}$), which returns a schema that covers the lost data and a mapping to populate that schema.

6.3 Merge

Now that we have new parts S'' of S', we need to combine it with $\mathcal{V}$. If S'' and $\mathcal{V}$ are expressed in the same metamodel, then this can be done with the Merge operation. It takes as input the two

[2] The term "tgd" comes from considering the mapping to be a logic formula of the form $\forall \bar{x}\ (\varphi(\bar{x}) \rightarrow \exists \bar{y}\ \psi(\bar{x}, \bar{y}))$, where $\bar{x}$ and $\bar{y}$ are sets of variables, $\varphi(\bar{x})$ is a conjunction of relation atoms, and $\psi(\bar{x}, \bar{y})$ is a conjunction of relation atoms that uses all of the variables of $\bar{y}$. So the tuples of relations in $\varphi(\bar{x})$ "generate" tuples of relations in $\psi(\bar{x}, \bar{y})$. If $\varphi(\bar{x})$ is restricted to use only relation atoms of the source schema, and $\psi(\bar{x}, \bar{y})$ is restricted to use only relation atoms of the target, then it is source-to-target.

schemas to be merged and a mapping between them that describes where the two schemas overlap. It returns a merged schema along with mappings between the merged schema and each of the two input schemas. In our example, the mapping required as input to Merge is calculated by composing the mappings on the path between S'' and $\mathcal{V}$. That is, $map_{V\text{-}S''} = map_{V\text{-}S'} \bullet map_{S'\text{-}S''}$.

If S'' is not expressed in the same metamodel as $\mathcal{V}$, then before doing the merge, we need to invoke the ModelGen operation on S'' to produce an equivalent schema S''' in $\mathcal{V}$'s metamodel and a mapping $map_{S''\text{-}S'''}$. Then we can merge S''' with $\mathcal{V}$ using $map_{V\text{-}S'''} = map_{V\text{-}S'} \bullet map_{S'\text{-}S''} \bullet map_{S''\text{-}S'''}$.

An instance-level semantics of Merge was given in [67]. Algorithms for computing a merged schema from input schemas when the input mapping is defined by exact match of element names appeared in [25] and when the input mapping is a set of correspondences in [67][82]. Some view integration algorithms can also be used as Merge algorithms [21][64][97]. Still, we are lacking an understanding of Merge relative to the expressiveness of its input and output mappings, as has been developed for Compose. Thus, when expressive mappings are used in schema evolution, new merge algorithms are likely to be needed.

6.4 Computing an Inverse

Suppose that database D is migrated to D′, as in Figure 5, but it is later determined that the migration was a mistake. If updates were applied to D′ after the migration, then the transformation $tran_{D\text{-}D'}$ from D to D′ needs to be reversed. That is, we need the inverse $tran_{D'\text{-}D}$ of $tran_{D\text{-}D'}$. This is not the same as the Invert operation of Section 6.2, which simply reverses the roles of the source and target of the mapping (which may be a relation, not a function). Rather, we need a transformation that can actually produce an instance D from an instance D′. Ideally, we would like this inverse to roundtrip. That is, given an instance D, if we use the forward transformation $tran_{D\text{-}D'}$ to produce D′ and then execute the inverse transformation $tran_{D'\text{-}D}$, we would like the result to be the same D that we started with. This is the same as the roundtripping condition described for ADO.NET in Section 4.

Fagin studies inverses of schema mappings in [37]. He formally defines the inverse of a mapping and gives several cases where it can be computed. In a follow-on paper [41], Fagin et al. introduce a relaxation of the notion of an inverse, called *quasi-inverse*. They give conditions where it does and does not exist and characterize the language to express inverses.

7. CONCLUSION

We have discussed a revised vision of model management—an infrastructure for tools that support data programmability. Model management operations include Match, ModelGen, TransGen, Compose, Diff, Merge and several others. The revised vision has two main aspects: first, the operations need to manipulate highly expressive mapping languages; and second, the runtime system to support mappings is part of model management. These aspects of the revised vision lead to many challenging research problems. We summarized recent work along these lines and highlighted some areas where additional work is most pressing.

The vision of model management is to encapsulate its operations in a schema and mapping manipulation engine that is used for a wide range of products where engineered mappings play a central role. To accomplish this, we need algorithms for all of the operations based on a common metamodel and expressive mapping language, and a way of using them to support metamodels and query languages that are in common use. Solutions to these problems are not in hand. We are still at the stage of reusing algorithms and designs for each new practical problem and mapping language that we face, not at the stage of reusing packaged components.

Schema mappings are proliferating. They are coming from ETL tools, object-to-relational mappers, report writers, and many other applications. Whether or not one buys the vision of model management, the need for more powerful and cost-effective solutions can hardly be denied. Thus, there are still many years of research ahead to greatly improve the quality of tool support we offer to help data architects solve the data programmability problems that arise from the design, implementation, and use of engineered mappings.

8. ACKNOWLEDGMENTS

We are grateful to José Blakeley, Christoph Freytag, Erhard Rahm, and Ivo Garcia dos Santos for many helpful comments.

9. REFERENCES

[1] S. Abiteboul and O.M. Duschka: Complexity of Answering Queries Using Materialized Views. *PODS 1998*: 254-263.

[2] S. Abiteboul et al.: The Lowell Database Research Self-Assessment. *Commun. ACM* 48(5): 111-118 (2005).

[3] S. Abiteboul, R. Hull, V. Vianu: Foundations of Databases. Addison-Wesley, 1995.

[4] A. Adya, J.A. Blakeley, S. Melnik, S. Muralidhar, and the ADO.NET Team: Anatomy of the ADO.NET Entity Framework, *SIGMOD 2007*.

[5] Altova, http://www.altova.com/

[6] P. Atzeni and R. Torlone: Management of Multiple Models in an Extensible Database Design Tool. *EDBT 1996*, 79-95.

[7] P. Atzeni, P. Cappellari and P. Bernstein: ModelGen: Model Independent Schema Translation. *EDBT 2006*, 368-385.

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

[9] BEA Aqualogic User Interaction, http://www.bea.com

[10] P.A. Bernstein: Applying Model Management to Classical Meta Data Problems. *CIDR 2003*.

[11] P.A. Bernstein, T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, D. Shutt: Microsoft Repository Version 2 and the Open Information Model. *Inf. Syst*. 24(2): 71-98 (1999).

[12] P.A. Bernstein, M.L. Brodie, S. Ceri, D.J. DeWitt, M.J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J.M. Hellerstein, H.V. Jagadish, M. Lesk, D. Maier, J.F. Naughton, H. Pirahesh, M. Stonebraker, J.D. Ullman: The Asilomar Report on Database Research. *SIGMOD Record* (27)4:74-80 (1998).

[13] P.A. Bernstein, Dayal, U., DeWitt, D.J., Gawlick, D., Gray, J., Jarke, M., Lindsay, B.G., Lockemann, P.C., Maier, D., Neuhold, E.J., Reuter, A., Rowe, L.A., Schek, H.-J., Schmidt, J.W., Schrefl, M., and Stonebraker: M. Future Directions in DBMS research—The Laguna Beach Participants. *SIGMOD Record* (18)1: 17-26 (1989).

[14] P.A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, I. Zaihrayeu: Data Management for Peer-to-Peer Computing : A Vision. *WebDB 2002*: 89-94.

[15] P.A. Bernstein, T.J. Green, S. Melnik, A. Nash: Implementing Mapping Composition. *VLDB 2006*: 55-66.

[16] P.A. Bernstein, L.M. Haas, M. Jarke, E. Rahm, G. Wiederhold: Panel: Is Generic Metadata Management Feasible? *VLDB 2000*: 660-662.

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

[18] P.A. Bernstein, S. Melnik, J.E. Churchill: Incremental Schema Matching. *VLDB 2006*: 1167-1170.

[19] P.A. Bernstein, S. Melnik, and P. Mork: Interactive Schema Translation with Instance-Level Mappings. *VLDB 2005*: 1283-1286.

[20] P.A. Bernstein and E. Rahm: Data Warehouse Scenarios for Model Management. *ER 2000*: 1-15.

[21] J. Biskup and B. Convent: A Formal View Integration Method. *SIGMOD 1986*: 398-407.

[22] J. A. Blakeley, D. Campbell, S. Muralidhar, A. Nori: The ADO.NET Entity Framework: Making the Conceptual Level Real. *SIGMOD Record* (35)4: 552-565 (2006).

[23] M.S. Bloor and J. Owen: Product Data Exchange. CRC Press, 1995.

[24] P. Bohannon, W. Fan, M. Flaster, P. Narayan: Information Preserving XML Schema Embedding. *VLDB 2005*: 85-96

[25] P. Buneman, S.B. Davidson, and A. Kosky: Theoretical Aspects of Schema Merging. *EDBT 1992*: 152-167.

[26] M. J. Carey: Data delivery in a Service-Oriented World: the BEA AquaLogic Data Services Platform. *SIGMOD 2006*: 695-705.

[27] T. Catarci and M. Lenzerini: Representing and Using Interschema Knowledge in Cooperative Information Systems. *Int. J. Cooperative Inf. Syst.* 2(4): 375-398 (1993).

[28] R.G.G. Cattell and D.K. Barry (editors) et al.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann, 2000.

[29] S. Chaudhuri and U. Dayal: An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record* 26(1): 65-74 (1997).

[30] L. Chiticariu and W.-C. Tan: Debugging Schema Mappings with Routes. *VLDB 2006*: 79-90.

[31] G.P. Copeland and D. Maier: Making Smalltalk a Database System. *SIGMOD 1984*: 316-325.

[32] Crystal Reports, http://www.businessobjects.com/products/reporting/crystalreports/default.asp.

[33] S.B. Davidson, G. Christian Overton, V. Tannen, L. Wong: BioKleisli: A Digital Library for Biomedical Researchers. *Int. J. on Digital Libraries* 1(1): 36-53 (1997).

[34] U. Dayal: Processing Queries Over Generalization Hierarchies in a Multidatabase System. *VLDB* 1983: 342-353.

[35] L. Donelson, P. Tarczy-Hornoch, P. Mork, C. Dolan, JA Mitchell, M. Barrier, H. Mei: The BioMediator System as a Data Integration Tool to Answer Diverse Biologic Queries. *Medinfo*: 768-72, 2003.

[36] ETL Tool Survey 2006-2007, http://www.etltool.com/

[37] R. Fagin: Inverting Schema Mappings. *PODS 2006*: 50-59.

[38] R. Fagin, P.G. Kolaitis, R.J. Miller, and L. Popa: Data Exchange: Semantics and Query Answering. *Theor. Comput. Sci.* 336(1): 89-124 (2005).

[39] R. Fagin, P.G. Kolaitis, and L. Popa: Data exchange: Getting to the Core. *ACM TODS* 30(1): 174-210 (2005).

[40] R. Fagin, P.G. Kolaitis, and L. Popa, W.C. Tan: Composing Schema Mappings: Second-order Dependencies to the Rescue. *ACM TODS* 30(4): 994-1055 (2005).

[41] R. Fagin, P.G. Kolaitis, L. Popa, and W.C. Tan: Quasi-inverses of Schema Mappings. *PODS 2007*.

[42] S.M. Falconer and M. Storey: Cognitive Support for Human-Guided Mapping Systems. Tech. Report DCS-318-IR, 2007, Univ. of Victoria , http://www.cs.uvic.ca/~seanf/files/cog_support_mapping_systems.pdf

[43] M.J. Franklin, A.Y. Halevy, and D. Maier: From Databases to Dataspaces: A New Abstraction for Information Management. *SIGMOD Record* 34(4): 27-33 (2005).

[44] M. Friedman, A.Y. Levy, and T.D. Millstein: Navigational Plans For Data Integration. *AAAI/IAAI 1999*: 67-73.

[45] A. Fuxman, M.A. Hernández, C.T.H. Ho, R.J. Miller, P. Papotti, and L. Popa: Nested Mappings: Schema Mapping Reloaded. *VLDB 2006*: 67-78.

[46] A. Gal: Managing Uncertainty in Schema Matching with Top-K Schema Mappings. *J. Data Semantics VI*: 90-114, Springer LNCS Vol. 4090/2006.

[47] M. Gubanov, P.A. Bernstein: Structural Text Search and Comparison using Automatically Extracted Schema. *WebDB 2006.*

[48] L.M. Haas: Beauty and the Beast: The Theory and Practice of Information Integration. *ICDT 2007*: 28-43.

[49] L.M. Haas, M.A. Hernández, H. Ho, L. Popa, and M. Roth: Clio Grows Up: From Research Prototype to Industrial Tool. *SIGMOD 2005*: 805-810.

[50] A.Y. Halevy: Answering Queries Using Views: A Survey. *VLDB J.* 10(4): 270-294 (2001).

[51] A.Y. Halevy, N. Ashish, D. Bitton, M.J. Carey, D. Draper, J. Pollock, A. Rosenthal, and Vishal Sikka: Enterprise Information Integration: Successes, Challenges and Controversies. *SIGMOD 2005*: 778-787.

[52] A.Y. Halevy, M.J. Franklin, and D. Maier: Principles of Dataspace Systems. *PODS 2006*: 1-9.

[53] A.Y. Halevy, Z.G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov: The Piazza Peer Data Management System. *IEEE Trans. Knowl. Data Eng.* 16(7): 787-798 (2004).

[54] Hibernate, http://www.hibernate.org

[55] R. Hull: Relative Information Capacity of Simple Relational Database Schemata. *SIAM J. Comput*. 15(3): 856-886 (1986).

[56] IBM FileNet Forms Manager, http://www.filenet.com/English/Products/Forms_Manager/.

[57] IBM Rational Data Architect, http://www-306.ibm.com/software/data/integration/rda/

[58] C. Keene: Data Services for Next-Generation SOAs. *SOA WebServices Journal*, 4(12), 2004. http://webservices.sys-con.com/read/47283.htm

[59] R. Kimball and J. Caserta. *The Data Warehouse ETL Toolkit*, Wiley and Sons, 2004.

[60] P.G. Kolaitis: Schema Mappings, Data Exchange, and Metadata Management. *PODS* 2005: 61-75.

[61] J. Lechtenbörger, G. Vossen: On the computation of Relational View Complements. *ACM TODS* 28(2): 175-208.

[62] F. Legler and F. Naumann: A Classification of Schema Mappings and Analysis of Mapping Tools. BTW 2007: 449-464.

[63] M. Lenzerini: Data Integration: A Theoretical Perspective. *PODS 2002*: 233-246.

[64] J. Lin and A.O. Mendelzon: Merging Databases Under Constraints. *Int. J. Cooperative Inf. Syst.* 7(1): 55-76 (1998).

[65] P. McBrien and A. Poulovassilis: A Uniform Approach to Inter-model Transformations. CAiSE 1999: 333-348.

[66] S. Melnik: *Generic Model Management: Concepts and Algorithms*, Springer LNCS 2967, 2004.

[67] S. Melnik, P.A. Bernstein, A.Y. Halevy, and E. Rahm: Supporting Executable Mappings in Model Management. *SIGMOD 2005*: 167-178.

[68] S. Melnik, P.A. Bernstein, A.Y. Halevy, and E. Rahm: A Semantics for Model Management Operators. MSR-TR-2004-59, http://research.microsoft.com, June 2004. An early but somewhat extended version of [67].

[69] S. Melnik, E. Rahm, P.A. Bernstein: Rondo: A Programming Platform for Generic Model Management. *SIGMOD 2003*: 193-204.

[70] S. Melnik, A. Adya and P.A. Bernstein, Compiling Mappings to Bridge Applications and Databases, *SIGMOD 2007*.

[71] Microsoft BizTalk, http://www.microsoft.com/biztalk/

[72] Microsoft Office InfoPath, http://office.microsoft.com/en-us/infopath

[73] Microsoft Sharepoint Server, http://www.microsoft.com /sharepoint

[74] Microsoft SQL Server Reporting Services, http://www.microsoft.com/sql/technologies/reporting/

[75] R.J. Miller, L.M. Haas, and M.A. Hernández: Schema Mapping as Query Discovery. *VLDB 2000*: 77-88.

[76] R.J. Miller, M.A. Hernández, L.M. Haas, L-L. Yan, H. Ho, R. Fagin, L. Popa: The Clio Project: Managing Heterogeneity. *SIGMOD Record* 30(1): 78-83 (2001).

[77] MSDN Library: The ADO.NET Entity Framework Overview. June 2006. http://msdn2.microsoft.com/en-us/library/aa697427(vs.80).aspx

[78] A. Nash, P.A. Bernstein, and S. Melnik: Composition of Mappings Given by Embedded Dependencies. *PODS 2005*: 172-183. Extended version to appear in *ACM TODS*.

[79] Oracle Toplink, http://www.oracle.com/technology/ products/ ias/ toplink/index.html

[80] OWL Web Ontology Language Reference, http://www.w3.org/TR/owl-ref/

[81] P. Papotti and R. Torlone: An Approach to Heterogeneous Data Translation based on XML Conversion. *CAiSE Workshops* (1) 2004: 7-19.

[82] R. Pottinger and P.A. Bernstein: Merging Models Based on Given Correspondences. *VLDB 2003*: 826-873.

[83] A. Poulovassilis, P. McBrien: A General Formal Framework for Schema Transformation. *Data Knowl. Eng.* 28(1): 47-71.

[84] Query Tools: Products, http://www.bitpipe.com/plist/term/Query-Tools.html

[85] Resource Description Framework, http://www.w3.org/RDF/

[86] E. Rahm and P.A. Bernstein: A Survey of Approaches to Automatic Schema Matching. *VLDB J.* 10(4):334-350 (2001).

[87] E. Rahm and P.A. Bernstein: An On-line Bibliography on Schema Evolution. *SIGMOD Record* 35(4):30-31, 2006. The full bibliography is at http://se-pubs.dbs.uni-leipzig.de/.

[88] G.G. Robertson, M. Czerwinski, and J.E. Churchill: Visualization of Mappings Between Schemas. CHI 2005: 431-439.

[89] M. Roth, M.A. Hernandez, P. Coulthard, L. Yan, L. Popa, H.C.-T. Ho, and C.C. Salter: XML Mapping Technology: Making Connections in an XML-centric World. *IBM Sys. J.* (45,2), 389-409 (2006).

[90] SAP Netweaver Portal, http://www.sap.com/usa/platform /netweaver/components/portal/index.epx

[91] Service Modeling Language, http://www.serviceml.org/

[92] P. Shvaiko and J. Euzenat: A Survey of Schema-based Matching Approaches. *J. Data Semantics IV*:146-171 (2005).

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

[94] A. Silberschatz, M. Stonebraker, and J.D. Ullman: Database systems: Achievements and opportunities. *Commun. ACM* (34)10: 110-120 (1991).

[95] J.M. Smith, P.A. Bernstein, U. Dayal, N. Goodman, T. Landers, K.W.T. Lin, E. Wong, "MULTIBASE -- Integrating Heterogeneous Distributed Database Systems," *Proc. of 1981 National Computer Conf.,* AFIPS Press, 487-499.

[96] Solidworks, http://www.solidworks.com/

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

[98] Stylus Studio, http://www.stylusstudio.com/

[99] Y. Velegrakis, R. J. Miller, and L. Popa: Mapping Adaptation under Evolving Schemas. *VLDB 2003*: 584-595.

[100] G. Wiederhold: Mediators in the Architecture of Future Information Systems. *IEEE Computer* 25(3): 38-49 (1992).

[101] Workshop on Information Integration, Oct. 2006, http://db.cis.upenn.edu/iiworkshop/index.htm

[102] L.-L. Yan, R.J. Miller, L.M. Haas, R. Fagin: Data-Driven Understanding and Refinement of Schema Mappings. *SIGMOD* 2001: 485-496.

[103] C. Yu and L. Popa: Semantic Adaptation of Schema Mappings when Schemas Evolve. *VLDB 2005*: 1006-1017.