Mappings as Building Blocks for Complex Integration

Lucian Popa
IBM Almaden Research Center
Outline

- **Schema mapping research: what can we do now**
  - Brief description of Clio

- **What could be next:**
  - Flows of mappings and transformations
  - How can we *increase automation* in creating such flows?
  - Mappings as *reusable* objects
  - Repository of common mappings and types

- **Disclaimer:**
  - this is a very "mapping-biased" look at integration 😊
  - there are plenty of important pieces left out ...

- **Result of several discussions with:** L. Haas, M. Hernandez, H. Ho, P. Kolaitis, R. Miller, M. Gubanov, H. Pirahesh, I.R. Stanoi, ...)
Schema Mappings (Recap)

- A schema mapping is a useful abstraction that specifies a relationship between a source schema and a target schema.
  - Can be used for: data transformation, query answering, to represent schema evolution, etc.

The power of the schema mapping concept relies in its *simplicity*.
- A set of **logical assertions** (e.g., dependencies) that can be manipulated.
- Independent of run-time engines (SQL, XQuery, ETL), but can be **compiled** into such engines.
Clio: From Mappings to Runtime Artifacts
A Detour into Clio
Clio: The System

Clio has a 3-level mapping “architecture”, first introduced in [PVMHF ’02]
- Correspondences
- Constraints
- Queries (Transformation)

Clio has two main generation components:
- (1) Correspondences are compiled into s-t constraints (tgds [BV’84])
  - Constraints represent the real mapping (used in all Clio related tools for generation, composition, evolution, etc.)
  - IBM mapping specification language (MSL, used in Rational Data Architect), derived from Clio specification
- (2) Constraints are then compiled into execution scripts (SQL, XSLT, ETL, Java, etc.)
Compilation of Correspondences into Constraints

\[ t_1: \text{Emp}(s,n) \land \text{WorkedOn}(s,p,d) \rightarrow \exists E \exists F ( \text{TEmp}(E,s,n) \land \text{EmpProjSummary}(E,p,d,F) ) \]

\[ t_2: \text{EmpEval}(s,n,p,f) \rightarrow \exists E \exists D ( \text{TEmp}(E,s,n) \land \text{EmpProjSummary}(E,p,D,f) ) \]

We can also write constraints as inclusion of CQs. For example, \( t_1 \):

\[
\begin{align*}
\text{select } e.ssn, e.name, w.pname, w.duration \\
\text{from } \text{Emp } e, \text{WorkedOn } w \\
\text{where } e.ssn = w.ssn
\end{align*}
\]

\[
\subseteq \begin{align*}
\text{select } e.ssn, e.name, s.pname, s.duration \\
\text{from } \text{TEmp } e, \text{EmpProjSummary } s \\
\text{where } e.eid = s.eid
\end{align*}
\]

- Source and target foreign key constraints are compiled into the mapping constraints
  - So, target constraints will be automatically satisfied
  - Data associations are preserved
- Some target fields are unspecified (e.g., eid)
  - But they will have to be generated (consistently)

\( \sum_{st} \)
Data Exchange: Theory

[FKMPO3]:
- Given source instance \( I \), and given schema mapping \( \Sigma_{st} \) (and possibly target constraints, \( \Sigma_t \)), what is the best target instance and how do we compute it?

Constraints underspecify the problem and there are multiple solutions (e.g., multiple ways of putting nulls, for example,)

Universal solutions are the most general solutions:
- E.g., never assume that two nulls are equal unless specified by the constraints

Chasing \( I \) with \( \Sigma_{st} \cup \Sigma_t \) yields a canonical universal solution \( J \)
- Populate the target with all the required tuples, adding fresh new nulls for the existential variables
- For termination, needs the constraints in \( \Sigma_t \) to be weakly acyclic
Data Exchange in Clio: Query Generation

- In Clio, target fk constraints are already taken into account
  - No need to chase them

- Clio implements universal solutions via query generation

- Main idea:
  - Skolemize and normalize the s-t tgds:
    - Skolemize and normalize the s-t tgds:
      - \[ \text{Emp}(s,n) \land \text{WorkedOn}(s,p,d) \rightarrow \text{TEmp}(E_1[s,n,p,d],s,n) \]
      - \[ \text{Emp}(s,n) \land \text{WorkedOn}(s,p,d) \rightarrow \text{EmpProjSummary}(E_1[s,n,p,d],p,d,F[s,n,p,d]) \]
      - \[ \text{EmpEval}(s,n,p,f) \rightarrow \text{TEmp}(E_2[s,n,p,f],s,n) \]
      - \[ \text{EmpEval}(s,n,p,f) \rightarrow \text{EmpProjSummary}(E_2[s,n,p,f],p,D[s,n,p,f],F) \]
  - We obtain single-headed rules (can also describe this as an SO tgd where \( E_1, E_2, D \) and \( F \) are existential functions)

- These are “GAV” mappings, so we can write SQL queries to populate the target.
The Rest of the Theory vs. Clio

**Other target constraints (non fks):**
- Target tgds (e.g., transitive closure), cyclic fks, target egds
- One problem: target query languages are not able, in general, to express chasing with such constraints
- Target egds (e.g., fds) may induce conflicts, which need to be resolved on a tuple by tuple basis, or by more specialized rules
- Better pushed into a separate phase

**Computing the core (smallest of the universal solutions)**
- Why would we do that? Less redundancy in the target data
- But again, cannot be compiled into a query (need specialized code)
- Better pushed into a separate phase
Composition
- Important (schema evolution, data flows)
- Clio implements composition of SO tgds (part of RDA too)

Data exchange theory is formulated in relational terms (to keep things clear!)
But in Clio, everything is extended to deal with XML too
So, a mapping can be used at design-time as an (indirect, declarative) model of the run-time execution.

But there are other uses of a mapping (in addition to being a model for execution) ...
Applications to Schema Evolution

But there are other uses of composition and inversion:

- store data with one mapping but query it using its inverse
- **reuse of mappings**: design a mapping between two logical schemas but deploy it between two physical schemas (**to come back at this**)

**Same vision here as the model management of Phil Bernstein**

- schemas and mappings are at its core
Schema Mappings: A summary of where we are

- We have developed **tools and methods** for:
  - mapping generation (including extensive work on schema matching),
  - mapping compilation into data transformation scripts
  - mapping-based query answering/rewriting, updates, invertibility
  - we are starting to make **small steps** towards interaction between the tools and the users
    - (e.g., visualization, debugging and explanation of mappings)

- We have also developed pieces of the **theory** (e.g., data exchange, query answering, composition, inversion)
So, what’s missing?

A mapping is often only one piece of a larger set of components (other mappings, transformations, black-box procedures) that need to be orchestrated together.

Why? Several reasons, but they mainly have to do with complexity of integration:

1) It may be easier to design a “flow” of small mappings that use intermediate results (“small” schemas) than a large complex mapping that goes directly from S to T.
   - The designer may not know what the target is or how to get there, so the transformation needs to be built incrementally in small steps
   - ETL and data mashup systems have the data flow flavor
     - but their level of abstraction is low (physical operators), with little opportunity for automation, optimization and reuse
2) The schemas S and T are complex but are built out of smaller modules (e.g., types). It may be easier to map the types individually.

- Need to assemble an umbrella mapping to invoke and correlate the smaller mappings (MapMerge)
- (And this could go on multiple levels, if we have complex types)

3) Finally, there may already exist other reusable mappings out there. One of them may not work but putting together several of them may do the job.
So, what’s missing?

- A system to support the *semi-automatic assembly of complex integration flows* (or large mappings) from existing small mappings.

- Sharing and reuse of mappings within different integration applications:
  - “Someone, somewhere, must have done something similar.”

- (General theme: *making our integration tools more usable and more modular*)
Mapping Reuse

First idea:
- There are not that many ways in which a piece of data can be transformed from a certain type to another type

Second idea:
- Applications in the same domain tend to use the same “common” types of data (although the concrete representations may vary)
  - Example: most integration scenarios in the HR domain will manipulate data about employees, jobs, departments, etc.
- Thus, there is a small set of common and frequently used types that are “buried” in a potentially large number of heterogeneous schemas
"manager-emp": example of a small reusable mapping between two common types
Mappings in Terms of Common Types

- If all our mappings are saved and stored in terms of these common types, then there is a significant potential for reuse
  - Relatively few common types (and mappings) but huge number of concrete data sources and schemas
  - We just need to “adapt” them to various, concrete, schemas
    - In the example, a mapping between the “common type” of manager and the “common type” of employee is reused between two concrete schemas that “contain” manager and employee

- One key issue: how do we come up with the common types?
  - Possibly, similar techniques used for schema integration,
  - There are already open source efforts towards standardizing (by popular consensus) the “common types”
    - Freebase (www.freebase.com)
Metadata Repository Challenges

 Metadata repository:
 - common types and their mappings,
 - concrete schemas and their mappings,
 - relationships between concrete schemas and common types

 Some of the issues:
 - Index/query the repository based on the common types
   - Find me all schemas about “employee”
 - Identify the “common types” in a concrete schema
   - What is this schema about?
   - May require matching
 - Search for similar schemas in the repository
 - Search for mappings or paths of mappings between a source
   concrete schema and a target concrete schema
   - By finding similar schemas that may have mappings, or
   - By identifying common types and using them as intermediate nodes
Example of Interaction with the Repository

Suppose we need a mapping (or flow) from $S$ to $T$

Alternative 1:
- Find schemas $S_1$, $T_1$ that are similar to $S$, $T$, respectively, and
- Find paths of mappings in the repository that “connect” $S_1$ and $T_1$
- Filter, compose, merge, union, or chose among the different mapping paths

Alternative 2:
- Find common types $O_1$, $O_k$ in schema $S$,
- Find common types $U_1$, $U_l$ in schema $T$
- Find mapping paths that connect $\{O_1, ..., O_k\}$ to $\{U_1, ..., U_l\}$
- Filter, compose, merge, union, or chose among the different mapping paths
- Note: merge is really needed here, since $O_1$, $O_k$ are only components in $S$ (similar for $T$ and its components)
Operations on Mappings Become Essential

- There is a small set of operations on mappings that are essential for the “assembly” process.

- Some of them are simple or relatively well-understood:
  - Union, filter, composition

- Other are relatively new or less understood:
  - Merge, inverse
  - Comparing alternative mapping paths (so that we can choose the “right” path)
More on MapMerge

Uncorrelated mappings (tgds):
- depts (d, dn) → ∃X tdepts(X, dn)
- emps (d, en) → ∃X (temps (X, en))

Better mappings:
- depts (d, dn) → ∃X tdepts(X, dn)
- depts (d, dn) ∧ emps (d, en) → ∃X (tdepts (X, dn) ∧ temps (X, en))

An even better mapping:
- depts (d, dn) → ∃X ( tdepts(X, dn) ∧ ( emps (d, en) → temps (X, en) )

“Dept” and “Emp” are the basic components in these schemas

Could be adapted from some generic dept→dept, emp→emp mappings

Put emp→emp mapping in the context by relating it to dept

Put emp→emp mapping in the context by making it a submapping of dept→dept

This is an example of a nested mapping [FHHMPP’06]
Concluding Remarks

- There are still many issues left unexplored on the semantics of mappings and their operations
  - Map merge may be more complex (may need user interaction)
  - What's the theory behind this?

- There is still little we know on comparing and selecting between alternative mappings (or mapping paths)
  - Visualization and browsing of the alternatives is essential

- There is significant potential for automation by enabling mapping reuse
  - Types and their mappings → finer granularity than monolithic schema mappings, better abstraction
  - Types are closer to application logic

- Flows are important