Integration Workbench: Integrating Schema Integration Tools

Peter Mork, Arnon Rosenthal, Len Seligman, Joel Korb, Ken Samuel
The MITRE Corporation
McLean, VA, USA
{pmork, arnie, seligman, jkorb, ksamuel}@mitre.org

Abstract

A key aspect of any data integration endeavor is establishing a transformation that translates instances of one or more source schemata into instances of a target schema. This schema integration task must be tackled regardless of the integration architecture or mapping formalism. In this paper we provide a task model for schema integration. We use this breakdown to motivate a workbench for schema integration in which multiple tools share a common knowledge repository.

In particular, the workbench facilitates the interoperation of research prototypes for schema matching (which automatically identify likely semantic correspondences) with commercial schema mapping tools (which help produce instance-level transformations). Currently, each of these tools provides its own ad hoc representation of schemata and mappings; combining these tools requires aligning these representations. The workbench provides a common representation so that these tools can more rapidly be combined.

1. Introduction

Schema integration is an integral aspect of any data integration endeavor. The goal of this paper is to organize the strategies and tools used in schema integration into a consistent framework. Based on this framework, we propose an open, extensible integration workbench to facilitate tool interoperation.

We view the development of a data integration solution to consist of three main steps: schema integration, instance integration and deployment. This paper focuses on schema integration, which generates a transformation that translates source instances into target instances.

This task involves first identifying, at a high level, the semantic correspondences between (at least) two schemata, a task we refer to as *schema matching*. Second, these correspondences are used to establish precise transformations that define a *schema mapping* from the source(s) to the target.

Researchers have built many systems to semi-automatically perform schema matching [1]. Schema mapping tools generally provide the user with a graphical interface in which lines connecting related entities and attributes can be annotated with functions or code to perform any necessary transformations. From these mappings, they synthesize transformations for entire databases or documents. These tools have been developed by commercial vendors (including Altova's MapForce, BEA's AquaLogic, and Stylus Studio's XQuery Mapper) and research projects (such as Clio [2], COMA++ [3] and the wrapper toolkit in TSIMMIS [4]).

Currently an integration engineer can choose to embrace a specific development environment. The engineer benefits from the automated support provided by that vendor, but cannot leverage new tools as they become available. The alternative is to splice together a number of tools, each of which has its own internal representation for schemata and mappings. In one case, we needed four different pieces of software to transform a mapping from one tool's representation into another.

By adopting an open, extensible workbench, integration engineers can more easily leverage automated tools as they become available and choose the best tool for the problem at hand.

1.1. Contributions

First, we describe pragmatic considerations that are important to the design of schema matchers: 1) contrary to conventional wisdom, many real-world schemata are well documented, so linguistic processing of text descriptions is important, 2) in several real-world scenarios, schema integration must be performed without the benefit of instance data, and 3) domain values are often available and could be better exploited by schema matchers.

Second, we establish a task model for schema integration based on a review of the literature and tools and on observations of engineers solving real-world integration problems. We presented our task model to three experienced integration engineers to verify that the model included all of the subtasks they had encountered.

The task model is important because it allows us to make comparisons: Among integration problems, we can ask which of the tasks are unnecessary because of simplifying conditions in the problem instance. Among tools, we can ask what each tool contributes to each task and quantify the impact in realistic settings.

Third, we describe how the task model and pragmatic considerations guide the development of a specific integration tool, in our case Harmony, a prototype schema matcher, which bundles a variety of match algorithms with a graphical user interface.

Our fourth contribution is to articulate the need for data integration among schema integration tools—our community can benefit in insight and utility by practicing what we preach. We propose a candidate collection of interfaces that constitute an integration workbench, which allows multiple integration tools to interoperate and provides a common knowledge repository for schemata and mappings. One outcome of the integration workbench is that integration engineers can more easily choose which match algorithms (or suites thereof [5]) to use when solving real integration problems. We offer this proposal as a discussion starter, which could ultimately lead to an open standard for interoperation of integration tools.

1.2. Outline

This paper is organized as follows: Section 2 contains our observations regarding schema integration efforts performed on behalf of the federal government. In Section 3 we describe a task model for integration problems. In Section 4 we present design desiderata based on the task model and describe how the Harmony schema matching tool addresses these desiderata. Finally, Section 5 describes the interfaces that constitute the integration workbench and Section 6 discusses future work.

2. Pragmatic Considerations

Conventional wisdom suggests that schema matching should focus on data instances because instances are common and documentation is sparse (or even incorrect). Whereas these phenomena may be observed in some domains, particularly web-based sources, it is often not the case for schemata developed for or by the US federal government (or, we suspect, other large enterprises).

From the perspective of an integration engineer, data instances may be extremely hard to obtain (the data exist, but are not available to the engineer) for at least two reasons.

- **Security/sensitivity**: Data instances are often more sensitive than their corresponding schemata—e.g., in defense applications, an integration engineer may have access to schemata but may lack sufficient clearances to access instances. Sometimes, an agency that owns the data is willing to share them with another agency, but not with the contracting integration engineers responsible for developing the initial mappings. Wider release of schema information is less problematic.
- **Conceptual schemata:** One may begin creating important mappings to and from a new system, even before it has any data or running applications. For example, the U.S. Federal Aviation Administration developed a mapping of some of its systems to a conceptual model for the new European Air Traffic Control System, before that system was implemented or had any instance data. As a general phenomenon, when one builds a data warehouse, the mappings from data sources are the actual means for populating it.

Thus, we have observed that it is not safe to assume the availability of instance data in enterprises. Instead, schema integration tools must use whatever information is available. Instance data, thesauri, etc. are sometimes available and sometimes not.

While instance data are often unavailable, we have found that many government (and probably many other enterprises') schemata are well documented. Evidence for this claim will now be presented.

We obtained a collection of 265 conceptual (ER) models from the Department of Defense metadata registry (which contains schemata only, no instances!). This repository contains 13,049 elements (entities or relationships) and 163,736 attributes. As indicated in Table 1, the vast majority of these items contain a definition of roughly one sentence.

This registry also explicitly enumerates domain values for which documentation is also available. For example, a schema for air traffic control introduces coding schemes for types of aircraft, runways and airports. Unfortunately, this documentation is often lost when a logical schema is converted into SQL. The standard approach is to store each coding scheme in its own relation, and each code as a

Table 1: Frequency and length of documentation in the DoD Metadata Registry

Item	Item Count	# With Definition	% With Definition	Word Count	Words/ Item	Words per Definition
Element	13,049	12,946	~99%	143,315	~11.0	~11.1
Attribute	163,736	135,686	~83%	2,228,691	~13.6	~16.4
Domain	282,331	282,128	~100%	1,036,822	~3.67	~3.68

string or integer value, *sans* documentation.

This approach is good for referential integrity, but bad for integration efforts. A better solution would be to define semantic domains for each coding scheme so that integration tools could more easily identify domain correspondences. In fact, when we asked integration engineers to describe how they approach an integration problem, a recurring pattern emerged. They first identified obvious top-level entity correspondences. But then, instead of proceeding to sub-elements or attributes, they then manually inspected the domain values to find correspondences. From this low-level, they then worked their way up the schema hierarchy to attributes, sub-elements, and finally back to top-level entities. Our task breakdown was designed to support this pattern.

3. Task Model for Data Integration

To better understand how schema integration tools assist an integration engineer, we enumerated the subtasks involved in schema integration. We started with a task model based on input from 147 survey participants familiar with schema integration from a research or practical perspective [6]. We extended that model to include the subtasks addressed by a variety of systems ([3, 4, 7-13]) and then presented it to three experienced integration engineers for validation. Based on their feedback, we extended the model to include subtasks not directly supported by any system.

At a high level, we consider 13 fine grained integration tasks, grouped into five phases: schema preparation, schema matching, schema mapping, instance integration and finally system implementation. During schema preparation, the source and target schemata are identified so that a set of correspondences can be identified during the matching phase. These semantic correspondences are formalized in the third phase as explicit logical mappings. Once schema integration is complete, instance integration reconciles any remaining discrepancies. In the final phase the integration solution is deployed.

In this section, we describe each phase in detail and describe how we evaluated the task model's completeness.

3.1. Schema Preparation

The first phase of schema (or data) integration captures knowledge about the source and target schemata, to facilitate the subsequent matching and mapping phases. It identifies the target schema, and organizes the source schemata. The specific subtasks are:

1) Obtain the source schemata. This step gathers available documentation and imports the source schemata into the integration platform. If the source schemata are not in a format compatible with the platform, this step also includes any necessary syntactic transformations.

2) Obtain or develop the target schema. If performed, this step is analogous to the previous step. In many cases, the target schema is defined by the problem specification (e.g., translate data into the following message format). In other cases, the target schema must be developed based on the queries to be supported, or to combine the data from multiple sources. This step is optional because the target schema may be derived from the correspondences identified among the source schemata, as is assumed in [8].

In both cases, one may enrich the schemata, e.g., by defining coding schemes as domains, or documenting constraints that are not documented in the actual system, either because the system does not support the needed constructs, or because nobody took the time to do so. Thus, the integration platform may enable richer descriptions than the underlying systems. One also needs a means to keep the metadata in synch, as the actual systems change.

3.2. Schema Matching

The second phase establishes high-level correspondences among schema elements. There is a semantic correspondence between two schema elements if instances of one schema element imply the existence of corresponding instances of the other [14].

If a target schema has been identified, these correspondences establish relationships between each source schema and the target. As noted in [8], in the absence of a target schema, correspondences can also be established between pairs of (or across sets of) source schemata.

For example, to publish data stored in a relational database into an XML message format, some correspondences indicate that tuples from the source relation will be used to generate XML elements. Additional correspondences indicate which attributes will be used to generate data values. For example, multiple relations might correspond to a single element because a join is needed to populate the element's attributes, or a single relation may correspond to multiple elements to match nesting present in the target.

3) Generate semantic correspondences. This step determines which schema elements loosely correspond to the same real world concepts. These correspondences establish a weak semantic link in that they indicate that instances of one element can be used to generate instances of the other.

Whereas this phase consists of a single step, we consider matching to be its own phase because of its importance and the research attention it has received. The exact transformations implied by a correspondence are detailed in the mapping phase.

3.3. Schema Mapping

The schema mapping phase establishes, at a logical level, the rules needed to transform instances of the source schemata into instances of the target. The mappings must generate results that adhere to the target schema (or the target must be modified to reflect accurately the transformed data).

The first four subtasks below establish piecemeal transformations, and are not performed in a particular order. Each transformation indicates the precise mechanism by which source data is used to generate target data. Note that at times these transformations cross the schema/instance boundary [15]. Once transformations have been established for each schema element, they are aggregated into a logical mapping and verified.

4) Develop domain transformations. For each pair of corresponding domains, a transformation must be developed that relates values from the source domain to values in the target domain. In the simplest case, there is a direct correspondence (i.e., no transformation is needed). However, it is often the case that an algorithmic transformation must be developed, for example, to convert from feet to meters, or from first- and last-name to full-name. In the most detailed case, the transformation can best be expressed using a lookup table (e.g., to convert from one coding scheme to a related coding scheme). Context mediation techniques can then be applied [16, 17].

5) Develop attribute transformations. The previous step handled the case where the same property was encoded using different domains. This step deals with properties that are different but derivable. Sometimes one provides a transformation from source to target values, either scalar (e.g., Age from Birthdate), or by aggregation (e.g., AverageSalaryByDepartment from Salary). Other transforms we have seen include pushing metadata down to data (e.g., to populate a type attribute or timestamp), and populating a comment (in the target) to store source attribute information that has no corresponding attribute.

6) Develop entity transformations. The next step is to determine the structural transformations necessary to generate instances of the target schema. In the simplest case, a direct 1:1 mapping can be established. Alternatively, multiple entities may need to be combined (e.g., using join or union) to generate a single target entity. Or, a single entity may need to be split into multiple entities (e.g., based on the value of some attribute), which effectively elevates data in the source to metadata in the target.

7) Determine object identity. For each entity in the target, the next step is to determine how unique identifiers will be generated. In the simplest case, explicit key attributes in the source can be used to generate key values in the target. This may include populating implicit keys (such as those inherited from a parent entity), or correctly establishing parent/child relationships in a nested metamodel. For arbitrarily assigned identifiers (such as internal object identifiers), Skolem functions are commonly employed (see, for example, [2]).

These four subtasks interact with schema matching because establishing transformations is an iterative process. For example, in the first pass, we might establish a transformation from Professor to Employee (since instances of the former are also instances of the latter). While working on the Course/Grade sub-schema, we might realize that, in some cases, Students are also Employees. This new insight requires us to refine the Employee mapping. In other words, the previously identified correspondences may be both imprecise and incomplete.

The remaining mapping subtasks produce an executable mapping.

8) Create logical mappings. The next step is to aggregate the piecemeal mappings, which all concerned individual elements, into an explicit mapping for entire databases or documents. Humans may need to specify additional information (e.g., to distinguish join from outerjoin) before automated tools can sew the pieces together. In most cases, this requires writing a query (over the source schemata) that generates instances of the target schema, although in the local-as-view formalism [18] the source schemata are expressed as views over the target schema.

9) Verify mappings against target schema. If the integration task included a specific target schema, the final step is to verify that the transformations are guaranteed to generate valid data instances (i.e., all constraints are satisfied). In some cases, the only solution may be to modify the target schema to reflect how it will be populated. If a target schema was not specified, the final step is to generate the target schema based on the logical mappings.

3.4. Instance Integration

At this point, the tasks involved in schema integration are complete, and we turn our attention to instance integration.

10) Link instance elements. Two instance elements (with different unique identifiers) may represent the same real-world object. This subtask merges these elements into a single element.

11) Clean the data. This subtask removes erroneous values from instance elements. A value may be erroneous because it violates a domain constraint or because it contradicts information from a more reliable source.

3.5. System Implementation

Finally we are ready to develop and deploy a system that addresses operational constraints—factors external to schema and instance elements. Examples include determining the frequency and granularity of updates and the policy that governs exceptional conditions.

12) Implement a solution. The integration system designed in this phase must address any operational constraints. The significance of these constraints on real-world integration systems was stressed by the integration engineers who reviewed the task model.

13) Deploy the application. This step does not receive much research attention, but ease of deployment is an important concern.

This task model guided our development of the Harmony schema matching tool.

4. Harmony

Harmony is a schema matching tool that combines multiple match algorithms with a graphical user interface for viewing and modifying the identified correspondences. The architecture for Harmony is shown in Figure 1. Harmony's contributions include adding linguistic processing of textual documentation to conventional schema match techniques, learning from the input of a human in the loop, and GUI support for removing clutter and iterative development, as discussed in following sections.

Harmony currently supports XML schemata, entity-relationship schemata from ERWin, a popular modeling tool, and will soon support relational schemata. Schemata are normalized into a canonical graph representation.

The Harmony match engine adopts a conventional schema integration architecture [5, 19-21]. It begins with linguistic preprocessing (e.g., tokenization, stop-word removal, and stemming) of element names and any associated documentation. Then, several *match voters* are invoked, each of which identifies correspondences using a different strategy. For example, one matcher compares the words appearing in the elements' definitions. Another matcher expands the elements' names using a thesaurus. For each [source element, target element] pair, each match voter establishes a confidence score in the range (–1, +1) where –1 indicates that there is definitely no correspondence, +1 indicates a definite correspondence and 0 indicates complete uncertainty.

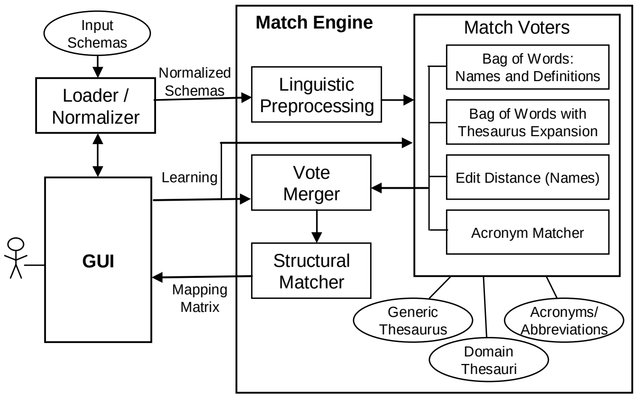

Figure 1: Architectural Overview of Harmony

Given *k* match voters, the vote merger combines the *k* values for each pair into a single confidence score. The vote merger weights each matcher's confidence based on its magnitude—a score close to 0 indicates that the match voter did not see enough evidence to make a strong prediction. The vote merger also weights each matcher *in toto* based on past performance (see Section 4.3).

A version of similarity flooding [22] adjusts the confidence scores based on structural information. Positive confidence scores propagate up the schema graph (e.g., from attributes to entities), and negative confidence scores trickle down the schema graph. Intuitively, two attributes are unlikely to match if their parent entities do not match.

Finally, these confidence scores are shown graphically as color-coded lines connecting source and target elements. The GUI provides various mechanisms for manipulating these lines, based on our design desiderata.

4.1. Design Goals

The considerations presented in Section 2 suggest that schema matching algorithms should not assume the absence of usable documentation. Many of the candidate matchers in the Harmony engine perform natural language processing and comparisons on this documentation. In our experience these matchers have good recall, although their precision is less impressive.

The task model in Section 3 suggests additional design desiderata. First, the integration engineer needs to be able to focus at different levels of granularity. For example, a common first step is to establish correspondences among conceptual sub-schemata. In the air traffic flow management domain, these sub-schemata might include facilities (airports and runways), weather, and routing. Note that the hierarchical and decomposable nature of XML Schema makes it easier to identify sub-schemata.

After establishing these high-level correspondences, the integration engineer focuses on one sub-schema at a time and delves into the details of the domains appearing in that sub-schema. The engineer wants to be distracted by neither correspondences pertaining to other sub-schemata nor those at intermediate levels of granularity.

A related goal is that the software tools must support iterative refinement. This desideratum is one of our motivations for developing the integration workbench described in Section 5. If data cannot flow freely among components, the engineer has little control over the order in which tasks will be completed.

The final desideratum is that all sub-tasks involved in schema integration must be supported. The commercially available tools naturally take this requirement more seriously than do research tools, such as Harmony. Whereas it is an interesting research problem to identify semantic correspondences, this contribution alone does not greatly assist the integration engineer. Because Harmony by itself does not currently support schema mapping, we defer further consideration of this desideratum to Section 5. We now consider how Harmony addresses the remaining desiderata.

4.2. Filtering

The Harmony GUI supports a variety of filters that help the integration engineer focus her attention. These filters are loosely categorized as link filters and node filters. A link filter is a predicate that is evaluated against each candidate correspondence to determine if it should be displayed. A node filter determines if a given schema element should be *enabled*. An enabled element is displayed along with its links; a disabled element is grayed out and its links are not displayed.

Harmony currently supports three link filters. First, a confidence slider filters links based on the confidence assigned to a link by the Harmony engine. Only links that exceed some threshold are displayed. Links that were drawn by the integration engineer, or were explicitly marked as correct, have a confidence score of +1. Similarly, links explicitly rejected have a score of –1.

The second filter determines if a link should be displayed based on whether it is human-generated or machine-suggested. The final filter displays, for each schema element, those links with maximal confidence (usually a single link, but ties are possible).

The node filters include a depth filter and a sub-tree filter. The former enables only those schema elements that appear at a given depth or above. For example, in an ER model, entities appear at level 1, while attributes are at level 2. Thus, using this filter, the engineer can focus exclusively on matching entities.

The sub-tree filter enables only those elements that appear in the indicated sub-tree. For example, this filter can be used to focus one's attention on the 'Facility' sub-schema. By combining these filters, the engineer can restrict her attention to the entities in a given sub-schema.

4.3. Iterative Development

Harmony supports iterative refinement through two mechanisms. First, the engineer can rerun the Harmony engine, which can learn from her feedback. Second, the engineer can mark sub-schemata as complete. We now describe these two mechanisms.

When the Harmony engine is invoked after some correspondences have been explicitly accepted or rejected (i.e., set to +1 or –1), this information is passed to the engine and used in two ways. First, each candidate matcher can learn from the user's choices and refine any internal parameters. For example, a bag-of-words matcher that weights each word based on inverted frequency increases or decreases word weight based on which words were most predictive. Second, the vote merger weights the candidate matchers based on their performance so far. Learning new weights must be done carefully, though. Each candidate matcher focuses on a particular form of evidence, such as elements' names. If the engineer based her first pass on exactly that form of evidence, the corresponding candidate matcher will appear overly successful.

In addition to accepting and rejecting specific links, the engineer can mark a sub-tree as complete. This action has several effects. First, it accepts every link pertaining to that sub-tree as accepted (if currently visible), or rejected (otherwise). Once a link has been accepted or rejected, the engine will not try to modify that link. This ensures that links do not mysteriously disappear or appear should the user subsequently invoke the Harmony engine.

Second, it updates a progress bar that tracks how close the engineer is to a complete set of correspondences. This feature was introduced at the request of integration engineers working on large schema integration problems that involve several dozen iterations.

Once all schema elements have been marked as complete, the final set of correspondences could be used to guide the generation of a more detailed mapping. Harmony provides neither a mechanism for authoring code snippets, nor a code generation feature; these would duplicate commercial capabilities. Instead, we are developing the integration workbench to couple our matching tools (and GUI) with commercially-available mapping products.

5. Integration Workbench

Our attempts to integrate Harmony with other schema integration tools revealed a key barrier to interoperability. Whereas schema integration experts trumpet the advantages of a modular, federated architecture that presents a unified view of multiple data sources, we have not applied that same insight when we develop our own systems. While some vendors may be moving in this direction internally to support integration of their own tools, they have not published their approaches or interfaces. There are obvious advantages to user organizations and small software companies to developing a *standard* framework for combining schema integration tools. We propose the following as a way to initiate discussion that could lead toward development of such a standard.

At the core of our workbench proposal is an integration blackboard, which is a shared knowledge repository. Mediating between the blackboard and the various schema integration tools is a workbench manager. The manager provides several services including transaction management, event services and query evaluation. The following sections describe the blackboard and manager.

5.1. Integration Blackboard

The integration blackboard (IB) is a shared repository for information relevant to schema integration that is intended to be accessed by multiple tools, including schemata, mappings, and their component elements. We propose using RDF [23] for the IB, because: 1) it is natural for representing labeled graphs, 2) one can use RDF Schema to define useful built-in link types while still offering easy extensibility, 3) it is vendor-independent, and 4) it has significant development support.

The basic contents of the IB are schema graphs and mapping matrices (an approach also taken in [19]). However, in RDF, any element can be annotated; we use this feature to enrich the graphs and matrices with additional information. We predefine certain annotations using a controlled vocabulary (these terms appear in sans serif).

5.1.1. Schemata: The IB represents a schema as a directed, labeled graph. The nodes of this graph correspond to schema elements. In the relational model, these elements include relations, attributes and keys. In XML, they include elements and attributes.

The edges of a schema graph correspond to structural relationships among the schema elements. These edges are object properties whose subject and object are both schema elements. For example, in the relational model contains-table edges are used to link a database to the tables it contains. Tables are linked to attributes via contains-attribute edges. In XML, elements are linked to subelements via contains-element edges, and to attributes via contains-attribute edges. For many schema languages, the edge-types are specified by the modeling language, but with ontologies they are extensible.

Whereas schema elements can be annotated arbitrarily, we identify three edge labels of particular importance to schema importing and matching utilities: name, type and documentation. Import tools populate these metadata so that they can be used by schema matchers to identify potential correspondences.

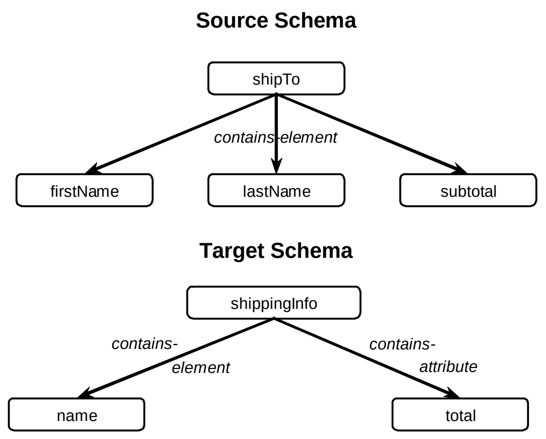

Figure 2: Sample schema graphs

code=
`let $shipto := $purchOrd/shipTo`
`return`
`  <shippingInfo total =`
`    "{ data($shipto/subtotal) * 1.05 }">`
`  {`
`  for $fName in $shipto/firstName,`
`      $lName in $shipto/lastName`
`  return`
`    <name>{`
`      concat($lName, concat(", ", $fName))`
`    }</name>`
`  }`
`  </ShippingInfo>`
	shippingInfo
	is-complete=false
	code=
		name
		is-complete=false
		code=
		`concat($lName,`
		`  concat(", ", $fName))`
			total
			is-complete=false
			code=
			`data($shipto/subtotal)`
			`  * 1.05`

shipTo
is-complete=false
variable=$shipto
	confidence=+0.8
	user-defined=false
		confidence=−0.4
		user-defined=false
			confidence=−0.6
			user-defined=false

firstName
is-complete=true
variable=$fname
	confidence=−1
	user-defined=true
		confidence=+1
		user-defined=true
			confidence=−1
			user-defined=true

lastName
is-complete=true
variable=$lname
	confidence=−1
	user-defined=true
		confidence=+1
		user-defined=true
			confidence=−1
			user-defined=true

subtotal
is-complete=true
variable=$shipto/subtotal
	confidence=−1
	user-defined=true
		confidence=−1
		user-defined=true
			confidence=+1
			user-defined=true

Figure 3: Sample mapping matrix in which every component has been annotated

Sample schema graphs appear in Figure 2. In the next section we present a sample mapping from the source schema to the target schema.

5.1.2. Mappings: Inter-schema relationships can be represented conceptually as a *mapping matrix*. This matrix consists of headers (describing source and target elements) plus content: a row for each source element and a column for each target element. (Note that whereas the structure can easily be interpreted as a matrix, we store this matrix using RDF.)

For example, the mapping matrix for the schemata in Figure 2 contains four rows and three columns, as shown in Figure 3. Each cell in the mapping matrix describes a potential correspondence between a source element and a target element.

Mapping elements are also annotated. First, each cell is annotated with confidence-score, which ranges from –1 (definitely not a match) to +1 (definitely a match), and is-user-defined. This latter annotation is true for any correspondence provided by the user (for example by drawing a link between two elements), and the associated confidence-score is ±1. When a match algorithm is executed, is-user-defined is false, and the confidence-score falls in the range (–1,+1).

Each row is further annotated with a variable-name. Each column is annotated with code that references these names. Finally, the matrix as a whole has a code annotation, which represents the mapping from source to target. Additional annotations are possible; for example, Harmony annotates rows and columns with is-complete to track progress. The relationship between these annotations and the mapping matrix appears in Figure 3.

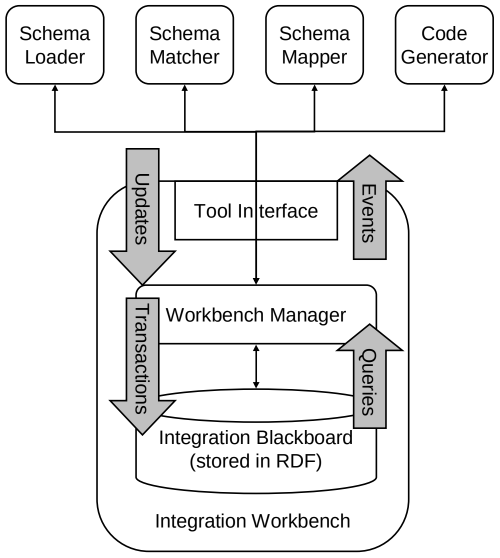

Figure 4: Workbench Architecture

5.1.3. Integration Blackboard Enhancements: We currently assume that the blackboard captures information about the source and target schemata, as well as the current state of the mapping that relates the source(s) to the target. Future goals include the following.

- The blackboard should maintain a library of mappings, partly to facilitate mapping reuse, but also as a resource for some matching tools.
- Schemata inevitably change; the blackboard should track schemata across versions.
- Mappings are also refined over time, especially once they are tested on real data. The blackboard should maintain mapping provenance.
- Based on Section 4.2, the blackboard should allow contextual information, such as focus on a particular subschema, to be shared across tools.
- The blackboard should be shared across multiple workbench instances.

5.2. Workbench Manager

All interaction with the IB occurs via the workbench manager, which coordinates matchers, mappers, importers, and other tools. The manager provides several services: First, it provides transactional updates to the IB. Second, following each update, it notifies the other tools using an event. Third, the manager processes ad hoc queries posed to the IB

A single-user version of the workbench architecture appears in Figure 4. Ultimately, we envision there to be one IB for each community of interest—i.e., a set of stakeholders "who must exchange information in pursuit of their shared goals, interests, missions, or business processes" [24]. Each integration engineer would have her own instance of the integration workbench containing a single manager and multiple tools.

5.2.1. Tools: We focus on four kinds of tools: loaders, matchers, mappers and code-generators. The first two tools support the first two phases of schema integration. Given the complexity of schema mapping, we separate out steps 4)–7), in which the mapping is produced piecemeal, from steps 8) and 9), in which code is generated.

Loaders are used during schema preparation to parse a schema from a file, database or metadata repository (including ancillary information such as definitions from a data dictionary) into the internal representation used by the IB. When the user invokes a loader, that tool places the new objects in the IB, which extends the mapping matrix accordingly and advises the other tools via an event.

Schema matching can be performed manually, as is the case for most commercial tools, or semi-automatically. (Harmony supports both approaches.) A match tool updates the cells of the mapping matrix. When correspondences are generated automatically, all of the interactions with the IB are wrapped in a transaction; no events are generated until the mapping matrix has been updated.

Schema mapping can also be performed manually or automatically [25], although we are not aware of any commercial automatic mapping tools. A mapping tool updates the code associated with each column. Both matchers and code generators may need to listen for these events to update their internal state.

Finally, a code-generator assembles the code associated with each column into a coherent whole. Thus, the code-generator must understand how to assemble code snippets based on the structure of the target schema graph (e.g., Clio [2]).

This enumeration of tools is by no means complete. Another tool might attempt to enforce domain-specific constraints on the mapping matrix. Or, a tool might annotate a schema with information culled from external documentation. All that is required is that a tool implements the tool interface.

The tool interface defines two methods. First, a tool must provide an invoke method. The implementation of this method might launch a GUI (for mapping), invoke a match algorithm, or display a file selection dialog (to load). Second, when the workbench starts, each tool has the option of implementing an initialize method. Generally, this is done when a tool needs to register for events.

5.2.2. Events: Tools generate events whenever they make any change to the contents of the IB. The workbench manager propagates these events to allow any tool to respond to the update. A different type of event is generated for each major component of the IB so that a tool can register for only those events relevant to that tool.

A schema loader generates a *schema-graph event* when it imports a schema into the workbench. Any tool with a GUI listens for these events and refreshes the display.

A *mapping-cell event* is generated when a user manually establishes a correspondence. Multiple such events are triggered by an automatic matching tool. A mapping tool can listen for these events to propose a candidate transformation, such as a type conversion.

Conversely, when a mapping tool establishes a transformation, it generates a *mapping-vector event*. Match tools listen for these events to synchronize the mapping cells with the updated row or column. A code generation tool similarly listens for these events to synchronize the assembled mapping. The code generation tool, in turn, generates a *mapping-matrix event* when the user manually modifies the final mapping.

Additional interactions are possible, but generally speaking, a tool listens for events immediately upstream or downstream in the task model. It is necessary to listen in both directions given the iterative behavior described in Section 4.3 and illustrated using a case study.

5.3. Case Study

We have begun validating the integration workbench by using it to allow Harmony and BEA's AquaLogic tool to interoperate. Both tools support schema loading and manual matching. Harmony also supports automated matching, but neither mapping nor code generation. Conversely, the AquaLogic development environment supports manual mapping and automatic code generation.

In our pilot study, AquaLogic is the first tool launched by the workbench. Within AquaLogic, the integration engineer can load schemata, connect source elements to target elements, and initiate the automatic generation of XQuery code. Alternatively, she can choose a sub-tree (including an entire schema) and request recommended matches from Harmony. The workbench launches the Harmony GUI and begins an IB transaction. The integration engineer uses Harmony to automatically propose likely correspondences, which she accepts or rejects using the GUI. Once satisfied, she exits Harmony to complete the IB transaction.

AquaLogic then updates its internal representation based on the changes made in Harmony. The integration engineer also provides element and attribute transformations that are incorporated into the generated XQuery. At any point this code can be tested on sample documents.

This combination of tools addresses all of the desiderata presented in Section 4.1. Harmony allows the integration engineer to focus on varying levels of granularity while matching, and AquaLogic supports all of the schema integration subtasks. Both tools support iterative refinement when used independently, as well as when combined. The next step will be to try the combined tool on real government schema integration problems.

6. Conclusions and Future Work

Data integration is a widely researched problem. However, we described ways in which enterprise data integration differs from the situations usually encountered in the research literature (e.g., documentation is widely available, instance data less so). Other pragmatic comments discussed how best to represent coding schemes so they can be leveraged by integration tools.

We also enumerated the subtasks involved in data integration, partitioned to reflect the behavior of integration engineers and the support provided by existing tools. This task analysis is intended to guide tool development and to enable comparisons across tools and integration problems.

Based on our observations and task modeling, we identified important design goals for integration tools. Specifically, we articulated the need to support all of the tasks involved in schema integration. One approach to meeting this need is to bring multiple tools to bear.

Unfortunately, assembling several tools to solve a particular integration problem is daunting. Our community needs to adopt the principle of assembling systems from modular components and integrating existing components.

To facilitate tool interoperation, we proposed an open, extensible integration workbench. This architecture provides a unified view of schemata and mappings so that integration tools can more easily communicate. We believe that both tool vendors and database researchers benefit from this arrangement. We hope that this proposal will generate discussion that ultimately could lead to standards (e.g., for mapping matrices) for data integration tool interoperation.

Since our overarching goal is to improve the lives of integration engineers, our next task is to perform a usability analysis of the Harmony/AquaLogic integration suite. We will measure the extent to which software tools save time on each of the schema integration subtasks.

7. Acknowledgements

We would like to thank Chris Wolf for his work developing the Harmony GUI, Mike Carey and Sachin Thatte for their input, and the anonymous reviewers for their feedback.

8. References

[1] E. Rahm and P. A. Bernstein, "A Survey of Approaches to Automatic Schema Matching," The VDLB Journal, vol. 10, pp. 334–350, 2001.

[2] R. Miller, M. A. Hernández, L. M. Haas, L. Yan, C. T. H. Ho, R. Fagin, and L. Popa, "The Clio Project: Managing Heterogeneity," SIGMOD Record, vol. 30, pp. 78–83, 2001.

[3] D. Aumueller, H. H. Do, S. Massmann, and E. Rahm, "Schema and ontology matching with COMA++," presented at Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, MD, 2005.

[4] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yerneni, M. M. Bruenig, and V. Vassalos, "Template-Based Wrappers in the TSIMMIS System," presented at Proceedings ACM SIGMOD International Conference on Management of Data, Tucson, AZ, 1997.

[5] A. Doan, P. Domingos, and A. Y. Halevy, "Learning to Match the Schemas of Databases: A Multistrategy Approach," Machine Learning, vol. 50, pp. 279–301, 2003.

[6] L. J. Seligman, A. Rosenthal, P. E. Lehner, and A. Smith, "Data Integration: Where Does the Time Go?," IEEE Database Engineering Bulletin, vol. 25, pp. 3–10, 2002.

[7] N. Ashish and C. A. Knoblock, "Wrapper Generation for Semi-structured Sources," SIGMOD Record, vol. 26, pp. 8–15, 1997.

[8] C. Batini, M. Lenzerini, and S. B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," ACM Computing Surveys, vol. 18, pp. 323–364, 1986.

[9] S. Cluet, C. Delobel, J. Siméon, and K. Smaga, "Your Mediators Need Data Conversion!," presented at SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, Seattle, WA, 1998.

[10] D. Florescu, A. Y. Levy, and A. O. Mendelzon, "Database Techniques for the World-Wide Web: A Survey," SIGMOD Record, vol. 27, pp. 59–74, 1998.

[11] A. Pan, J. Raposo, M. Álvarez, J. Hidalgo, and Á. Viña, "Semi-Automatic Wrapper Generation for Commercial Web Sources," presented at Engineering Information Systems in the Internet Context, Kanazawa, Japan, 2002.

[12] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. D. Ullman, "A Query Translation Scheme for Rapid Implementation of Wrappers," presented at Deductive and Object-Oriented Databases, Fourth International Conference, DOOD'95, Singapore, 1995.

[13] L. Popa, Y. Velegrakis, R. Miller, M. A. Hernández, and R. Fagin, "Translating Web Data," presented at VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, 2002.

[14] R. Fagin, P. Kolaitis, R. Miller, and L. Popa, "Data Exchange: Semantics and Query Answering," presented at Database Theory — ICDT 2003, 9th International Conference, Siena, Italy, 2003.

[15] C. M. Wyss and E. L. Robertson, "Relational Languages for Metadata Integration," ACM Transactions on Database Systems, vol. 30, pp. 624–660, 2005.

[16] C. H. Goh, S. Bressan, S. E. Madnick, and M. Siegel, "Context Interchange: New Features and Formalisms for the Intelligent Integration of Information," ACM Transactions on Information Systems, vol. 17, pp. 270–293, 1999.

[17] E. Sciore, M. Siegel, and A. Rosenthal, "Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems," ACM Transactions on Database Systems, vol. 19, pp. 254–290, 1994.

[18] J. D. Ullman, "Information Integration Using Logical Views," presented at Database Theory—ICDT '97, 6th International Conference, Delphi, Greece, 1997.

[19] P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix, "Industrial-Strength Schema Matching," SIGMOD Record, vol. 33, pp. 38–43, 2004.

[20] H. H. Do and E. Rahm, "COMA - A System for Flexible Combination of Schema Matching Approaches," presented at VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China, 2002.

[21] J. Madhavan, P. A. Bernstein, and E. Rahm, "Generic Schema Matching with Cupid," presented at VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, Roma, Italy, 2001.

[22] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity Flooding: A Versatile Graph Matching Algorithm," presented at Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, 2002.

[23] D. Brickley and R. Guha, "RDF Vocabulary Description Language 1.0: RDF Schema," World Wide Web Consortium (W3C®), 2003. http://www.w3.org/TR/rdf-schema/

[24] J. P. Stenbit, "Department of Defense Net-Centric Data Strategy," 2003. http://www.defenselink.mil/nii/org/cio/doc/Net-Centric-Data-Strategy-2003-05-092.pdf

[25] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga, "CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies," presented at Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, 2004.