AL$MONY: Exploring Semantically-Assisted Matching in an XQuery-Based Data Mapping Tool

M. Carey[1], S. Ghandeharizadeh[2], K. Mehta[1], P. Mork[3], L. Seligman[3], S. Thatte[1]

[1]BEA Systems
2315 North First Street
San Jose, California 95131
405-570-5363
{mcarey,kattul,sachin}@bea.com

[2]Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
213-740-4781
shahram@usc.edu

[3]The MITRE Corporation
7515 Colshire Drive
McLean, VA 22101
703-983-1465
{pmork,seligman}@mitre.org

ABSTRACT

In this paper, we describe an in-progress effort to augment the data integration capabilities of a service-oriented data integration product with the advanced matching capabilities of a knowledge-based schema matching workbench, thereby providing a guided, interactive, and "what"-oriented design environment for use by data architects and integration engineers. The data integration product is the BEA AquaLogic Data Services Platform, which provides a declarative, XML-based foundation for integrating and service-enabling heterogeneous enterprise data sources in order to deliver data services for use by SOA applications. The schema matching tool is the Harmony system from MITRE, a state-of-the-art prototype system that maintains a growing knowledge base of matching information and employs a composite matching architecture beneath a rich UI to guide the ranking and decision-making processes involved in schema matching. We motivate this work, review the capabilities of each of the systems, and sketch the architectural approach that we are taking to combining the two systems into a synergistic whole.

1. INTRODUCTION

A growing challenge faced by many enterprises today is how to access and analyze data residing in different sources such as database management systems (DBMSs), legacy systems, Web services, and XML files to name a few. This is due to three forms of heterogeneity, see Figure 1. The first, commonly termed physical heterogeneity, is due to the fact that different sources have different APIs and data formats. For example, an RDBMS source may provide a JDBC or ODBC interface, a file system a bytestream API to access XML files in unparsed character form, and a Web service source might be accessed using JAXRPC. Once this form of heterogeneity is resolved, the formal representation used by the alternative systems to describe their data might be different, sometimes termed logical heterogeneity. For example, one system may provide a relational representation while another provides an XML one. Most often the participating systems also exhibit differences in conceptual organization of data even though they pertain to the same application. This is commonly termed semantic heterogeneity and occurs when independent designers model the requirements of the same application. There are several reasons for having semantic as well as conceptual heterogeneity, ranging from enterprises acquiring one another to evolution of business units that merge after having developed systems that address their individual data needs [6].

A data integration system should address all three forms of heterogeneity of its target data sources. By doing so, it becomes a middleware layer that understands the conceptual, logical and physical organization of data in each participating data source. Its output is an integrated view that then pertains to an abstraction that truly serves the needs of the enterprise.

The framework that empowers an integration engineer to author the desired abstraction layer is the focus of this paper. The ideal

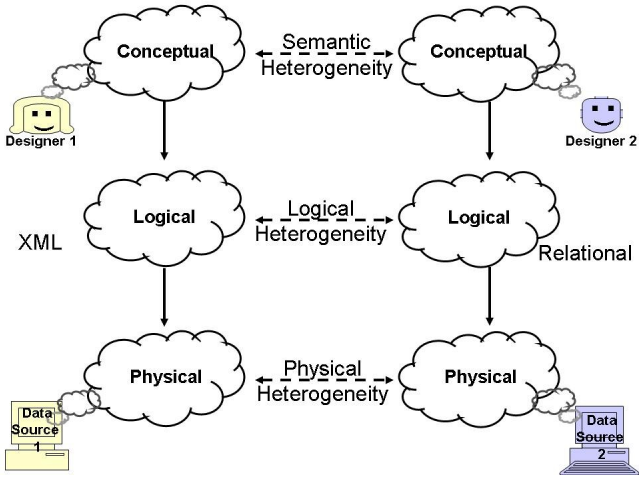

Figure 1: Semantic, logical and physical heterogeneity.

International Workshop on Semantic Data and Service Integration (SDSI'07), September 23, 2007, Vienna, Austria.

framework would be "what"-oriented, requiring the engineer to specify little more than the desired final abstraction and the participating data sources. In response, the framework determines the different forms of heterogeneity and how they are to be resolved. This is as compared to a "how"-oriented framework that requires the engineer to specify how the system should resolve heterogeneity. For example, BEA's ALDSP 2.5 [2, 3] system provides a "what"-oriented interface to handling physical and logical heterogeneity. Its approach to semantic heterogeneity is "how"-oriented, however, because the engineer is required to have a priori knowledge of all data sources and to author the mapping expressions that instruct the system on how to integrate the data sources to resolve semantic heterogeneity.

A "what"-oriented framework is desirable for two reasons. First, it minimizes the amount of data source specific (e.g., relational DBMS such as Oracle and IBM's DB2, WSDL of W3C for Web services) knowledge required from the engineer to integrate data, enabling the engineer to focus on the final requirements of the enterprise. Second, at times, the final applications dictating the abstraction desired from a data integration effort are fluid and their requirements change continuously. This is particularly true during the early phases of an integration project life-cycle. A "what"-oriented framework empowers the engineer to rapidly change the integrated abstractions to focus on application changes.

To realize a more "what"-oriented framework, we propose to build on the strengths of two complementary systems: BEA's AquaLogic Data Services Platform (ALDSP) [2, 3] and MITRE's Harmony [9, 10]. ALDSP employs a declarative foundation (not unlike [13]) to enable a user to design, develop, deploy, and maintain a framework that understands both the logical and semantic heterogeneity of data sources. Harmony automatically suggests, at a high level, possible semantic correspondences between (at least) two schemata and provides an intuitive GUI to enable a user to refine those correspondences. The resulting combined system is the AquaLogic data services platform integrated with harMONY, AL$MONY.

The AL$MONY design decisions described here are based on a combination of our joint work, our experience with current BEA customers who have been developing services using ALDSP, and Harmony case studies [9]. In terms of joint work, we have exchanged ideas and software and have begun the process of defining and prototyping the relevant inter-system APIs. This workshop paper therefore describes the design of AL$MONY based on our progress thus far and our plans. The rest of the paper is organized as follows. Sections 2 and 3 provide overviews of ALDSP and Harmony, respectively. Section 4 describes the AL$MONY vision and how it synergistically integrates the best of ALDSP and Harmony. Section 5 concludes the paper.

2. AquaLogic Data Services Platform

The AquaLogic Data Services Platform (ALDSP) provides a service-oriented integrated view of enterprise data sources. A given view of enterprise data, termed a dataspace, consists of a set of interrelated data services. A data service has a "shape", describing its information content. It may have a set of read functions, which are service calls that provide various ways to request access to one or more instances of the data service's business objects. In addition, a data service may include procedures that change its instances, i.e., insert, delete, and update. Moreover, a data service may have navigation functions to traverse relationships from one instance from one data service (e.g., Customer) to one or more instances of another (e.g., Order). ALDSP represents this shape information using XML Schema. Functions and procedures are represented as XQuery functions that can be invoked via Java calls, via Web services, in queries, and/or used to create other user defined data services.

Figure 2 provides a high-level summary of the ALDSP approach to data services. The bottom right consists of a physical collection of data sources external to ALDSP. These might include a mix of databases, Web services, packaged applications, legacy mainframe data, and other data sources. When an integration engineer points ALDSP to an enterprise data source, ALDSP introspects the source's metadata; it does so, for example, by obtaining the SQL metadata for a relational data source or processing the WSDL file for a Web service. This introspection guides the automatic creation of one or more *physical data services* that make the data source available for use in the XML world of ALDSP. Applying this process to a relational data source produces one data service per table or view. The return shape in this case is the natural XML representation of the table or view schema. In the presence of discoverable foreign key constraints, ALDSP also produces navigation functions across different data services. ALDSP introspects a Web service by processing its WSDL to yield one data service per distinct Web service operation return type. Multiple Web service operations might have the same return type and are modeled as multiple functions for the data service. The input signature of each function will be dictated by the input of the operation as specified by the WSDL. Other functional data sources are similarly modeled. The result, referred to as the physical model in Figure 2, is a uniform, "everything is a data service" view of the available data sources.

The center of Figure 2 shows the view of data seen by client applications. Instead of interfacing with the underlying data sources directly, or even with the more uniform physical model, an ALDSP client application sees a meaningful set of interrelated logical data services. Each such data service is an abstraction of' some coarse-grained entity that was chosen to be meaningful to the integration engineer and to serve as a reasonable unit of interaction for client-server calls. Each logical data service is internally composed from one or more data services of the physical model (or logical data services belonging to lower abstraction levels) using XQuery as the service composition language. Data services belonging to lower abstraction levels appear to the composer as functions that consume and produce simple- and complex-typed XML data. XQuery is a declarative,

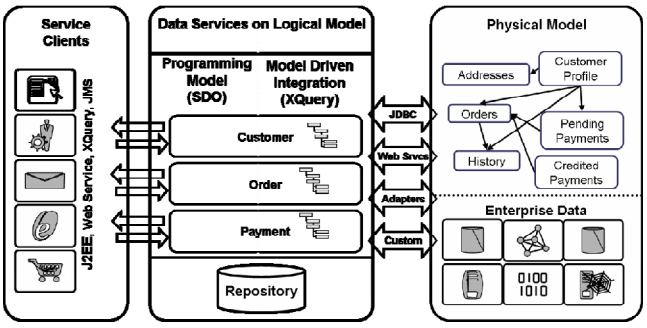

Figure 2: Overview of ALDSP

functional language well-suited to orchestrating such functions and also to transforming XML data in order to produce the desired business object shapes out of the information accessible from the underlying services.

2.1 An Example of How to Build a Data Service Using XQuery

To illustrate how one would use ALDSP to build a data service, consider an integration engineer who wishes to build a logical data service to retrieve customer profile information. Assume the relevant information resides in several different physical data sources. Suppose one relational DBMS, say Oracle, contains CUSTOMER and ORDER tables, while another DBMS, perhaps DB2, contains customer CREDIT_CARD information. These data sources are kept separate for security and auditing purposes. In addition to these relational data sources, assume a Web service provides a credit rating operation named getRating. The input to getRating is the customer's name and social security number. Its output is an integer value representing the credit rating for the input customer. The engineer registers these with ALDSP by simply pointing it to each physical data source.

Next, the engineer may author a function named getProfiles( ) to retrieve all instance of the integrated view. Figure 3 shows a screen-shot of the XQuery design view of ALDSP while performing this task. Its right side shows the desired integrated view. (Section 2.2 describes how the engineer may construct this target view using either a top-down or a bottom-up approach.) An instance of the target will consist of one customer, nested order data, nested credit card data, and a credit rating for a given customer. The engineer has already identified the relevant data sources by dragging them onto the canvas, see Figure 3. In addition, the engineer has employed this interface to map elements of the physical data sources to those of the target schema.

Further examination of Figure 3 reveals that the engineer has correlated the different physical data sources with one another by specifying join conditions between them. For example, the engineer has specified joins between the CUSTOMER and CREDIT_CARD data sources using their CID element.

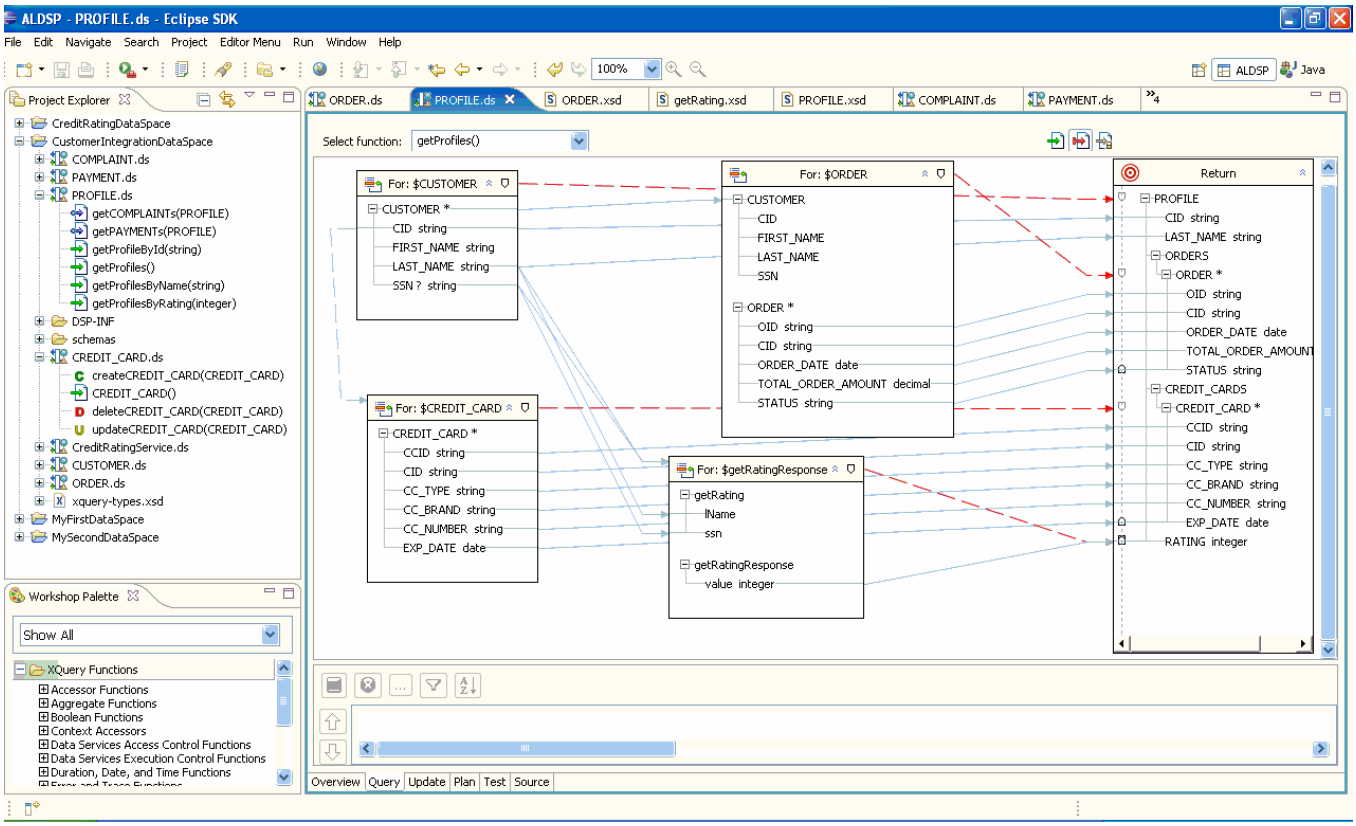

Figure 3: XQuery design view.

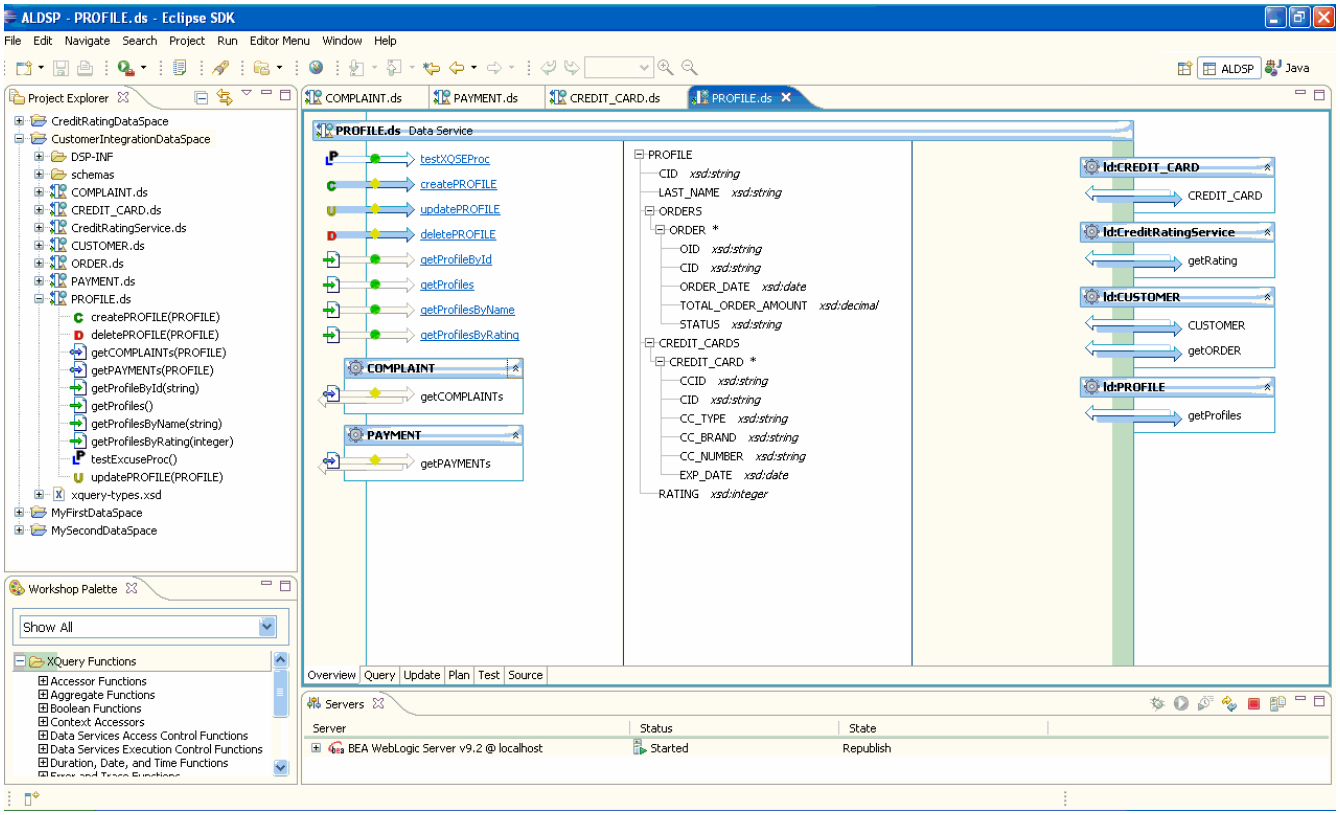

Figure 4: PROFILE data service.

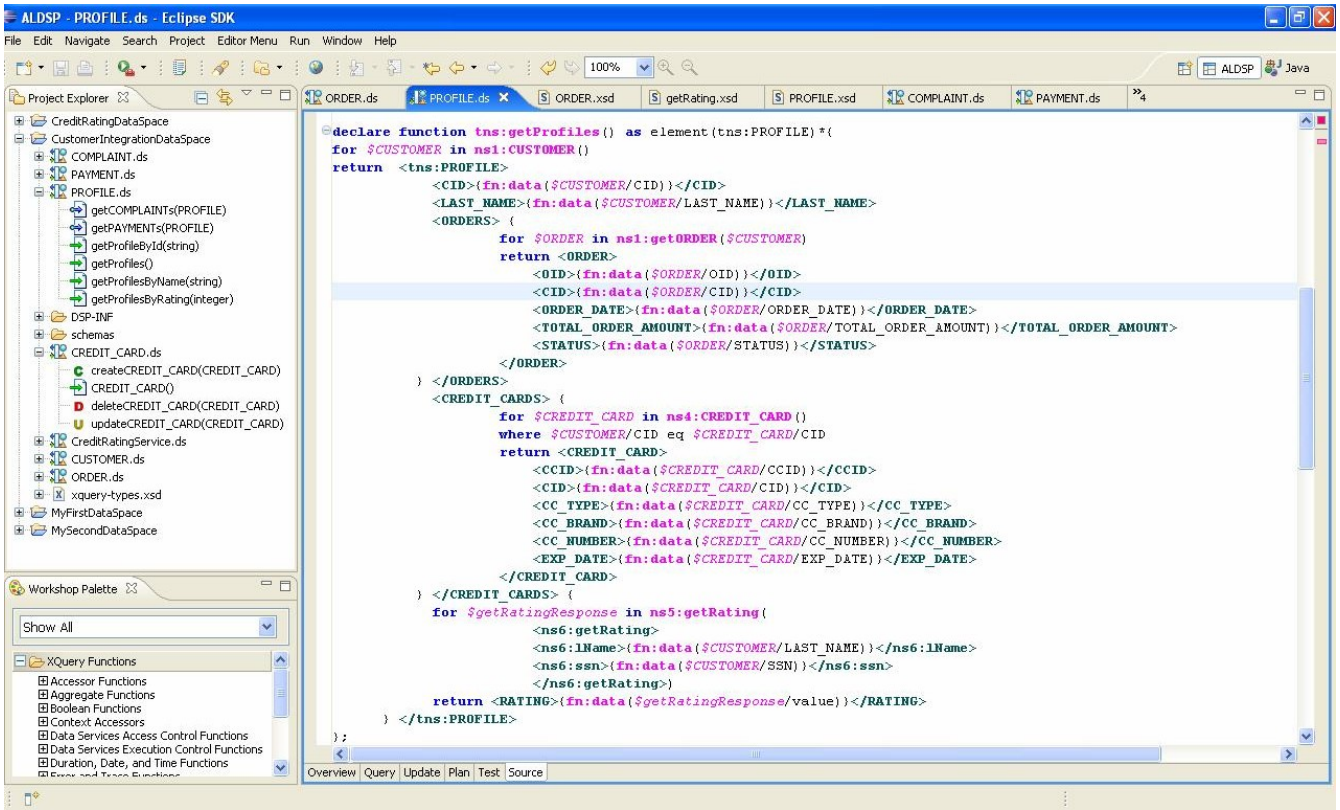

Figure 5: XQuery source view.

Figure 4 shows the different functions and procedures specified by the engineer on the PROFILE data service, such as functions getProfilesByName( ) to obtain the integrated profiles for all customers with a specific name, getProfilesByID( ) to retrieve the profile for a customer by customer id, and getProfilesByRating( ) to get profiles for all customers with a specified credit rating. Specification of these functions is trivial; they can be defined as selections using getProfiles( ), which has encapsulated all of the integration and correlation details for the customer profile data service as a "get all instances" function. This "integrate once and reuse" design pattern for data services is extremely important as it is the key to achieving data independence in ALDSP.

The lower left side of Figure 4 lists the navigation functions, i.e., getCOMPLAINTs( ) to retrieve instances from the COMPLAINT data service and getPAYMENTs( ) to retrieve instances of PAYMENTS data service pertaining to a given customer profile instance. Also, note the presence in the figure of the procedures createPROFILE, updatePROFILE, and deletePROFILE which create, update, and delete instances of PROFILE. Finally, the right side of Figure 4 shows the lower-level data services that constitute the PROFILE data service.

Figure 5 shows the actual XQuery source that ALDSP generates for one of the data service functions specified on the PROFILE data service (namely the function from Figure 3). This method getProfiles( ) takes no arguments and is responsible for computing and returning the complete profiles for all customers that appear in the CUSTOMER table of the first relational database. An XQuery function provides access to the table contents in the body of the query. For each customer returned by that function, getProfiles( ) accesses the ORDER table to create the nested <ORDERS> element. In this example, this access is via a navigation function that ALDSP automatically created for this purpose based on the foreign key constraint of the ORDERS table on the CUSTOMER table. Also for each customer, the CREDIT_CARD table in the second DBMS is accessed, and finally the getRating operation of the credit rating Web service is invoked. The result of getProfiles( ) is a series of <PROFILE> elements, one per CUSTOMER, which integrates all of this information from the different data sources. The specification shown in Figure 5 is optimized by ALDSP prior to being executed [2].

2.2 ALDSP Lifecycle

As should be evident from the example, ALDSP assumes the engineer has knowledge of the desired target view and how the physical data sources match and map both with the target schema and with one another. This might have been specified by the requirements of an application and its desired level of abstraction.

The engineer may populate the desired target schema by taking either a top-down or bottom-up approach. With a top-down approach, the engineer represents the target schema in XSD and imports it into ALDSP's data service view to construct the shape of the target data service, see Figure 3. With a bottom-up approach, the engineer again employs the XQuery design view, but in this case specifying no initial shape for the target. This results in an empty "return" schema. Next, the engineer registers the various physical data sources and then incrementally incorporates their schemas into the overall "return" schema for the data service as a side effect of the editing operations involved in building the first read function for the data service.

With both top-down and bottom-up approaches, ALDSP assumes that the engineer has a priori knowledge of the relevant data services, elements of source data services that are equivalent to a desired target data element, and the transformations (if any are needed) to map these elements. For large target schemas requiring data from numerous data sources, the integration effort may become difficult, motivating the need for a tool such as Harmony to assist the engineer with the matching process. Such a tool is especially useful when the engineer's a priori knowledge of physical and target data services is lacking.

3. Harmony

In the preceding section, we explained how the ALDSP IDE provides a convenient GUI for incrementally collecting, from the integration engineer, a set of known schema correspondences and mapping them into an executable XQuery function. In this section, we describe Harmony [9], a collection of tools that help a user to quickly identify the relevant schema correspondences. Armed with these correspondences, the integration engineer could more efficiently use ALDSP to author an XQuery function.

We begin this section by describing the Harmony match engine that generates, for each possible correspondence, a confidence score that indicates the extent to which a pair of schema elements appear to match. The match engine includes a suite of match voters, each of which considers a particular body of evidence. A vote merger then combines the information provided by the voters to generate a single score. Unlike other schema matching tools, the Harmony match engine considers both the quality of the available evidence and the quantity of that evidence.

Next we cover the Harmony GUI, which provides an intuitive interface for the integration engineer to indicate which of the possible schema correspondences are correct, and which are not. Possible correspondences indicated by the match engine are color-coded based on the confidence score. The GUI also provides a number of filters that allow the integration engineer to focus his attention. For example, the integration engineer can suppress all of the possible correspondences for which the confidence score is beneath some threshold.

We then describe how an integration engineer would use Harmony (in its current form) as part of an integration project. Based on our experiences using Harmony, we realized it is not trivial to couple Harmony with other schema integration tools. This realization inspired the development of the Harmony integration workbench originally described in [9]. Section 4 describes our current effort to use this workbench to implement AL$MONY, the integration of ALDSP with Harmony.

3.1 Harmony Match Engine Overview

The architecture for Harmony is shown in Figure 6. As indicated, it operates as follows. First, schemas are loaded and normalized into a canonical graph-based representation, and then the names and text definitions associated with schema elements undergo linguistic preprocessing (including tokenization, stemming, and stopword removal).The Harmony match engine is (in the parlance of [11]) a composite matcher, in that several *match voters* are run in parallel, and these results are combined by a *vote merger*. Each match voter considers some source of evidence to generate a match score for a particular (source element, target element) pair. Current match voters include *bag-of-words*, in which each element name and its definition is considered a "document" and the distance between each source and target element is determined using standard information retrieval techniques, *bag-of-words with thesaurus expansion*, *edit distance* of element names, and an *acronym matcher*. There is a built-in generic thesaurus (Roget); however, domain-specific thesauri and acronym dictionaries can also be used.

The standard approach for arriving at confidence scores is to compute the ratio of positive evidence to total evidence. However, this approach ignores the fact that, for a given evidence ratio, more evidence is better than less evidence. Moreover, by considering the amount of evidence observed by each match voter, the vote merger can more flexibly combine match scores.

In the Harmony match engine, match scores range from –1 to +1; intuitively, a match score of 0 indicates that, based on the evidence, the likelihood of a match is impossible to determine. As the ratio of positive evidence to total evidence increases, the match score should increase. For a fixed ratio, as the total evidence increases, the match score should also increase.

Based on this intuition, we can establish some theoretic bounds. If there is an infinite amount of positive evidence, the match score should equal +1. However, if there is no positive evidence, but an infinite amount of negative evidence, the match score should equal –1. Finally, if there no evidence (of either type), the match score should be 0.

Using these bounds, in [10] we show how to compute such a match score. For a given (source element, target element) pair, let *poe* represent the amount of positive observed evidence, and *toe* represent the total observed evidence. Given these values, the match score (*ms*) is defined as follows.

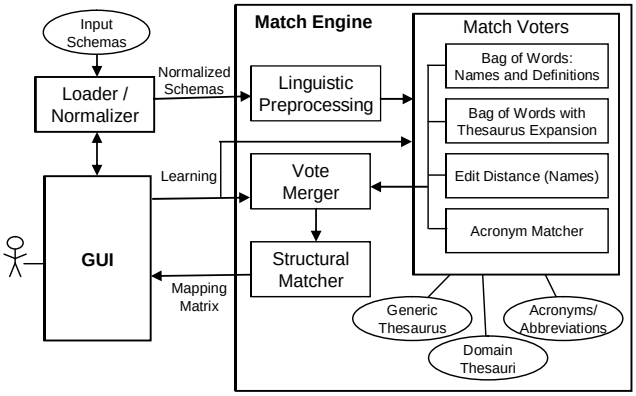

Figure 6: Architectural overview of Harmony.

$$ms = \ln\left(\frac{\left(\frac{x + k \times poe}{1 + k \times toe}\right)^{1/j}(e-1)+1}{(1 + x + k \times poe)^{1/(x+k\times poe)}}\right)$$

The numerator in this equation is based on the ratio of positive to total observed evidence. The denominator in this equation is a scaling factor based on the amount of positive observed evidence. Each match voter measures the observed evidence differently; in [10], we also describe a way to quantify evidence based on the documentation used to describe schema elements.

The vote merger is responsible for combining multiple match scores into a single confidence value. This combination is based on multiple factors including the score (*ms*) generated by a given match voter, the amount of evidence available to that match voter (*ef*), and the weight assigned to that match voter. For each (source element, target element) pair, the match voter generates a single confidence value.

The basic vote merging algorithm is a weighted average of the match scores generated by each match voter. If we assume that each match voter *v* is weighted equally, then the final confidence score (*conf*) is defined as follows, where *V* is the set of all match voters.

$$conf = \frac{\sum_{v \in V} ef \times ms_v}{\sum_{v \in V} ef}$$

In general, *ef* needs to range from zero (indicating the absence of evidence), to one (given infinite evidence). Thus, any function that maps *toe* to the interval [0, 1] fulfills this condition. Note, though, that the match score is close to zero when there is little total evidence, and approaches ±1 as the total observed evidence approaches infinity. Based on this observation, Harmony uses the absolute value of the match score as *ef*.

$$conf = \frac{\sum_{v \in V} |ms_v| \times ms_v}{\sum_{v \in V} |ms_v|}$$

Intuitively, this equation weights the match score generated by each match voter based on the magnitude of that score. This simplification works because a match score of zero indicates insufficient evidence to determine if the source element and target element match. A score close to ±1 indicates strong evidence either in support of a match, or against a match.

The Harmony match engine iterates over every possible (source element, target element) pair and generates a confidence score. The integration engineer can visualize these values using the Harmony GUI.

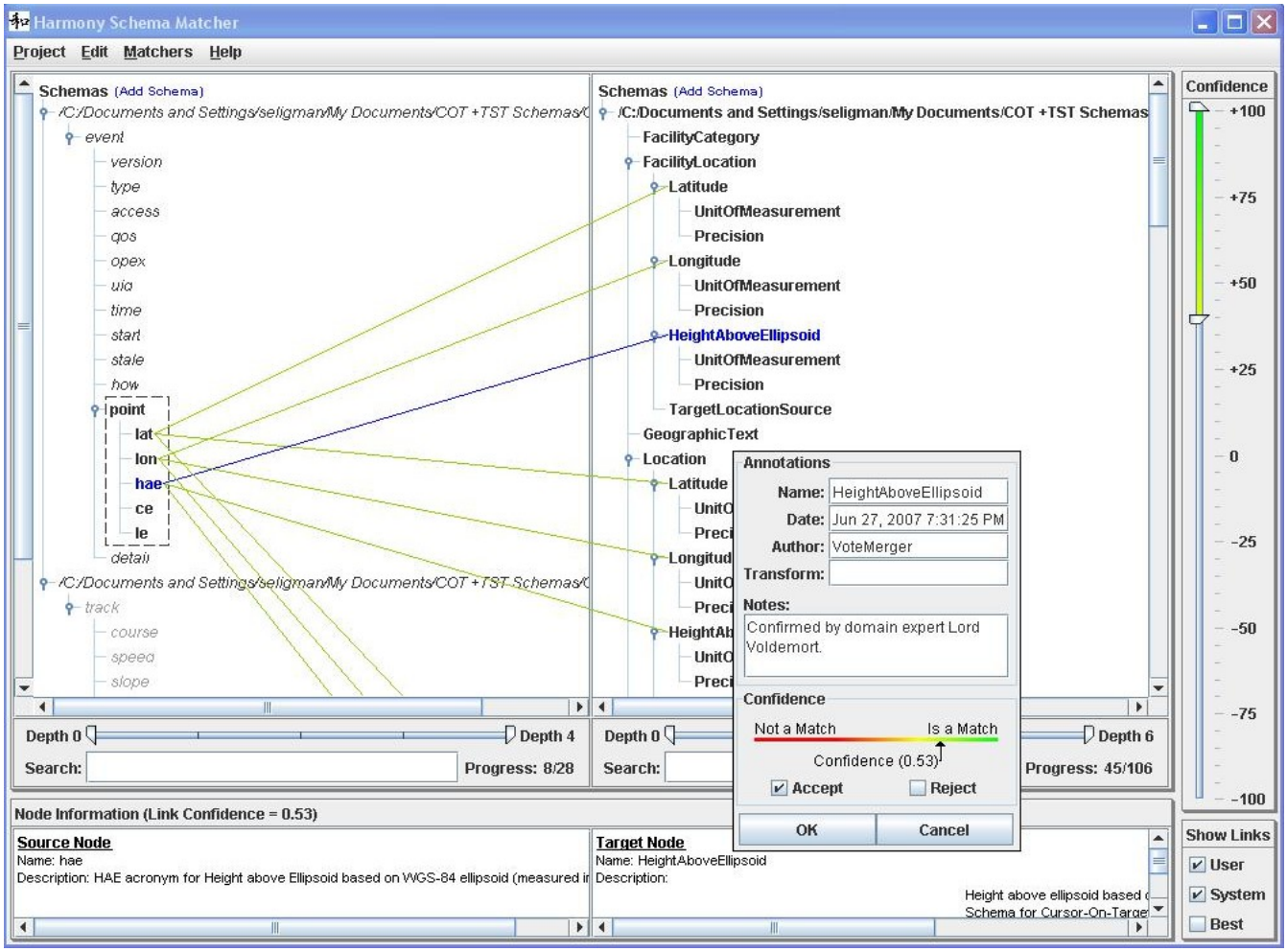

Figure 7: Harmony provides several mechanisms for focusing the integration engineer's attention.

3.2 Harmony GUI Overview

Like many similar products, the Harmony GUI displays the source schema on the left side of the screen, and the target schema on the right side of the screen (see Figure 7). Potential correspondences are shown as lines (links) connecting a source element to a target element. Harmony color-codes these links based on the confidence score associated with that link. Green links indicate large positive confidence scores, yellow links indicate mediocre positive confidence scores, and red links indicate negative confidence scores.

Using the GUI, the integration engineer can accept and reject potential links. He can also manually draw lines between a source and target element to indicate a true correspondence.

The Harmony GUI supports a variety of filters that help the integration engineer focus his 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 different link filters, all of which show up as options on the right side of Figure 7. First, a confidence slider filters links based on the confidence assigned to a link by the Harmony match engine. Only links that exceed the specified 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 that have been explicitly rejected are given a score of –1.

The second link filter determines if a link should be displayed based on whether it is human-generated or machine-suggested. The final link filter displays, for each source and/or target schema element, only those links whose confidence score is greater than all other links for either the source schema element or the target.

For node filters, Harmony includes 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-schema. In Figure 7, a sub-tree filter has been applied to the source schema on the left. By combining this filter with the previous, the engineer can restrict his attention to the entities in a given sub-schema.

The Harmony GUI also supports iterative refinement through two mechanisms: When the match 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 match engine and used in two ways. First, each match voter can learn from the user's choices and refine any internal parameters [10]. Second, the Harmony vote merger weights the match voters based on their performance so far.

In addition to accepting and rejecting specific links, the engineer can mark a sub-tree as complete. This action has two 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 match engine will not modify that link. This ensures that links do not mysteriously disappear or appear should the user subsequently invoke the Harmony engine. Second, the action updates a progress bar that tracks how close the engineer is to having 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. In the next section, we describe how an integration engineer uses the match engine and GUI to solve data integration problems.

3.3 Harmony Lifecycle

In [9], we introduced a task model for data integration. At a high level, we consider 17 integration tasks, grouped into five phases: schema preparation, schema matching, schema mapping, instance integration and finally system implementation. During the schema preparation phase, the source and target schemata are identified and transformed to a canonical model so that 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.

The Harmony system targets the first two phases: schema preparation and schema matching. (ALDSP supports end-to-end integration by providing tools for schema preparation, schema mapping and system implementation.) Harmony supports schema preparation by providing schema importers for common formats such as XML Schema, RDF and OWL. In addition, the schemata can be augmented by introducing project-specific thesauri and acronym dictionaries.

Once the integration engineer imports the source and target schemata, the GUI displays these on the left and right sides of the screen. The engineer can then manually identify semantic correspondences or run the Harmony match engine. Most of the engineer's time is spent deleting spurious matches and manually drawing links between elements missed by the match engine.

Eventually, the integration engineer reaches a point wherein all of the source and target schema elements have been matched (or marked as complete *sans* match). At this point, the engineer can export the project in several formats such as an Excel spreadsheet. The default format is to save the project as an RDF file in which each resource is a link between a source schema element and a target schema element. Because this file is plain RDF, it can be manipulated programmatically using an RDF query language.

Unfortunately, it is still difficult to import this format into another integration tool (such as ALDSP). More generally, we have yet to observe a pair of integration tools from different vendors that can seamlessly interoperate. For a simple integration task involving only a dozen schema elements, we found that we needed to use five separate tools, with custom glue between each pair (!).

This experience led us to consider creating a generic integration workbench that would allow heterogeneous data integration tools to communicate with one another via a shared blackboard and messaging framework. As a result, we introduced just such an integration framework in [9], which we summarize in the next section.

3.4 Integration Workbench

Our initial attempts to integrate Harmony with other schema integration tools revealed a key barrier to tool interoperability. While we as 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 developing our own systems. While some vendors may be moving in this direction internally, to support interoperability of their own tools, they have not published their approaches or interfaces. There would be obvious benefits to user organizations and small software companies to the development of a *standard* framework for combining schema integration tools.

At the core of the Harmony workbench design 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 workbench manager provides several shared services, including transaction management, event services, and query evaluation. The following sections describe the integration blackboard and the workbench manager.

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 [3] 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 (and growing) development support.

The basic contents of the IB are schema graphs and mapping matrices. 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; new integration tools can add others as required.

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 sub-elements 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.

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. 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). Each row of the matrix is further annotated with a variable-name, and each column with code snippets that reference these variables. Finally, the matrix as a whole has a code annotation, which assembles the code snippets from the individual cells into an executable whole. The ability to attach code and other annotations makes mapping matrices a generalization of similarity matrices [1, 5]. While similarity matrices share knowledge only among multiple matchers, mapping matrices can be used by matchers, mappers, and code generators.

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. The architecture for the integration workbench appears in Figure 8.

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.

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 be performed manually or automatically as well [7], at least in principle, although we are unaware of any commercial automatic schema 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.

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

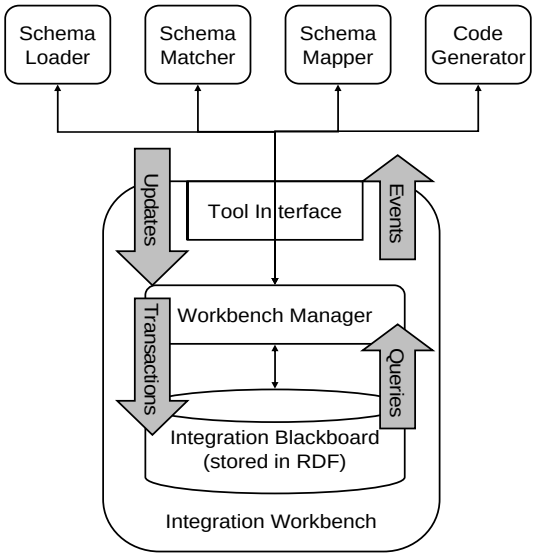

Figure 8: Integration workbench architecture.

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.

The RDF blackboard and event model allow multiple integration tools to interoperate. In the next section, we describe our initial efforts to demonstrate this assertion by integrating Harmony and ALDSP.

4. AL$MONY

The goal of the AL$MONY (pronounced “all money”) project, short for AquaLogic data services platform integrated with harMONY, is to combine the strengths of ALDSP and Harmony to provide an interactive user experience that facilitates the “easy” creation of new data services from a large and possibly choppy sea of enterprise data sources together with other, existing data services that may already have been created. As such, AL$MONY aims to simplify the matching and mapping of:

1) Source data services to a target schema. E.g., map the CID element from the CUSTOMER source to the CID element of the data service target schema in Figure 3.

2) Function outputs from available source data services to function inputs (and/or to a target schema). E.g., map the LAST_NAME and SSN elements from the CUSTOMER source to the lName and ssn input document schema of the getRating( ) Web service source call in Figure 3, and map the result of the Web service call into the target schema of the data service being defined.

3) Available data services together, to combine them via joins or unions. E.g., create a join predicate between CID from CUSTOMER and CID from CREDIT_CARD in Figure 3.

AL$MONY is designed to empower users to match and map data rapidly and interactively starting with much less knowledge than ALDSP requires today. Today, to build an integrated data service, an ALDSP integration engineer must come armed with knowledge of what/where the relevant data sources are and a

clear understanding of exactly how their schemas and their instances need to be matched and mapped. If the integration engineer lacks such in-depth, a priori knowledge of the data sources or how they should be integrated, AL$MONY should empower them to query and browse the metadata of the target schema and the available data services to obtain the knowledge needed to integrate them successfully. The kinds of questions posed by the user for a matching and mapping task may include:

- For a new (or ongoing) data service integration project, what data services appear likely to be of interest?
- For a given data service that has been determined to be of interest, what other data services appear likely to be of interest as well?
- For a given element or attribute within an identified set of interesting data services, what matches it within that set and should therefore be considered for mapping?

These general questions cover a variety of more detailed, context-specific questions that arise during the data service authoring process. For example, when matching and mapping multiple data services to the target schema of a new data service under construction, the developer may wish to inquire: "I have as my design goal the creation of a data service with target schema T – so what existing data services have relevant matching attributes?" Parts of this question are covered by the first two questions above. Effective user interfaces will clearly be a key component for enabling such context-specific questions to be asked in a useable, transparent way. These may include wizards that guide the developer through a set of question/answer screens specific to a context (in this case source selection)

4.1 AL$MONY Lifecycle

A key challenge, both from usability and architectural angles, is that AL$MONY must integrate the current Harmony and ALDSP lifecycles into a single, more unified user experience. We plan to realize this in two ways. Today, the Harmony model has a rather "waterfall"-like nature (e.g., matching precedes mapping), while the ALDSP data service editing UI is more interactive and incremental but presumes perfect a priori knowledge. To couple the two effectively, we will utilize the event model of the Harmony workbench to populate its blackboard from an ALDSP project. We will also integrate key features and ideas from the Harmony user interface with those of the ALDSP XQuery editor. Scale is another challenge, both in the number and sizes of data source schemas. ALDSP and Harmony have both focused to date more on "in the small" rather than "in the large" versions of the bulleted questions above, but we would like AL$MONY to eventually be capable of scaling up to the whole-enterprise level (and beyond). On the user interface side, techniques such as those presented in [12] offer a good start. We intend to build upon those ideas and explore others as well.

As an example of lifecycle integration, when an integration engineer asks AL$MONY to enable access to a new data source such as a relational database management system, AL$MONY will employ ALDSP's metadata import facilities to also populate the integration blackboard (IB) of Harmony. This should be relatively simple because ALDSP represents its data sources in XML Schema form, and transformation of XML Schema into Harmony's directed, labeled graph model already exists; see the loader discussion of Section 3.4. Similarly, when the engineer drags such a source onto the query canvas, a Harmony event will be generated to add the source's metadata to the current working set of schemas for its matching algorithms to operate on.

4.2 AL$MONY User Experience

Extending the graphical XQuery editor of ALDSP with the display-tailoring capabilities of Harmony's user interface will yield a much more effective user experience for the integration engineer whose a priori knowledge is imperfect. When the engineer constructs an integrated target view in a top-down manner (see Section 2.2), after registering the shape of the target data service, the engineer should then be able to inquire about other data services that potentially relate to the elements of the target schema. In response, AL$MONY can invoke Harmony to obtain match suggestions with confidence scores and process them to determine the data services most likely to be relevant to the target data service. The color codes of Harmony can be used to highlight the different sources in the XQuery graphical view: green for positive, yellow for mediocre positive, and red for negative confidence scores.

Once the set of suggested data sources has been provided, the engineer can employ AL$MONY's graphical cues to choose the relevant data sources and drag-and-drop them onto the canvas of XQuery's graphical view. Graphical selection of an attribute of a data service (either source or target) should then also trigger AL$MONY to identify the potential matching elements of the displayed data services. Once again, Harmony's color coding can be used to highlight these elements based on the computed confidence scores. The engineer can then use Harmony-style link and node filters to reduce screen clutter and explore the most relevant matches, per the discussion of Section 3.2, within the XQuery graphical mapping interface.

It is important to remember that the integration engineer is the sole authority whose specified matches are final. In particular, matches will be finalized implicitly in AL$MONY through the creation of the concrete XQuery mapping expressions that the engineer specifies via their graphical editing work. AL$MONY must then support behavior similar to Harmony's GUI when the engineer accepts or rejects specific link matches. For this reason, the engineer's accept and reject decisions, inferred from their XQuery editing actions, will lead AL$MONY to generate events for the workbench manager of Harmony. Since XQuery editing is an interactive and incremental process, and since data services can also evolve further over time as things in the enterprise change and require them to be changed as well, AL$MONY must support an incrementally changing knowledge base of matching information.

An interesting issue in the combined interface, which targets the interactive production of mappings as opposed to "just" matches, has to do with the role and the impact of type information. The ALDSP XQuery editor attempts to keep the engineer on a path where there is always a syntactically complete and fully type-correct XQuery underlying the screen state. Matches that involve differently typed source and target elements or attributes could be used in AL$MONY to auto-suggest required type-conversions and related computations. The acceptance of such matches and mappings by the engineer can be remembered as a future hint that, at least in the context of the schemas currently involved, this

particular type mismatch should not be interpreted by Harmony's type-based matching module as a negative contribution to the likelihood of the validity of an overall match. More broadly, the question arises of how the availability of the detailed mapping expressions might be most effectively exploited to glean and retain knowledge for use in future matching runs.

4.3 AL$MONY Architecture

The Harmony integration workbench includes a rich API for interrogating the blackboard and for interacting with other pluggable integration tools. To handle the ALDSP/Harmony system interactions needed for the scenario above, we have created a simplified "matcher SPI" (service provider interface) for the ALDSP IDE to use to invoke the features of Harmony or possibly other matching service engines. Thus far this API enables ALDSP to ask Harmony to a) introduce manually identified matches into its IB and b) access the matching knowledge base and identify potential new semantic correspondences among the current set of schemas. Using the existing Harmony workbench API, we were able to implement these two SPI methods with minimal code. The first method essentially required three lines: 1) iterate over the set of manually identified matches, 2) lookup the corresponding cell of the mapping matrix, and 3) update the confidence score for that cell. The second required four lines: 1) invoke the Harmony match engine, 2) identify the modified cells, 3) iterate over these cells and 4) add the potential match to the result. While these interactions between Harmony and ALDSP are limited, the evidence so far suggests that the two integration tools can be very effectively integrated, with minimal effort, via the integration workbench.

Ideally, AL$MONY should provide a plug-n-play framework so that it can be customized with knowledge libraries useful for different application domains. Harmony already supports the import of domain-specific thesauri to be used to detect additional candidate matches (e.g., that in the human resources domain, "employee" might correspond to "worker"). Beyond simple thesaurus-based term expansion, we would also like to support more full-featured ontologies such as UMLS, a widely used ontology of biomedical and health-related terms and their inter-relationships. Harmony's match algorithms could be extended to use "semantic distance" between terms as evidence in assigning scores for candidate matches. AL$MONY's GUI must support ontology export as well as import. Export is useful when the user's interaction with AL$MONY results in changes to a well-established ontology such as UMLS. The user may then share their exported ontology with other colleagues who are utilizing AL$MONY for data integration activities in the same general domain.

4.4 Moving Beyond Mapping

Last but not least, our long-term vision for the AL$MONY effort extends beyond ALDSP/Harmony integration. A subsequent stop on the road from "in the small" to "in the large" exploitation of schema matching and semantic user assistance could involve the injection of Harmony-style matching capabilities into enterprise repository products such as the ALDSP Enterprise Repository, ALER [14]. ALER today is capable of providing metadata management functions for a variety of software assets, ranging from business processes and Web services to other common components and shared services (even whole applications). ALER records and maps the relationships and interdependencies that connect software assets in order to promote reuse, improve impact analysis, and so on. A key aspect of reuse is discovery, and in the world of SOA, many of the same source identification issues that we described for ALDSP appear, but on a larger scale, in the context of service and component discovery in an enterprise repository.

A move toward integration in the large presents both challenges and opportunities. Current match algorithms do not scale to support discovery in large enterprise repositories such as the U.S. Department of Defense Metadata Registry, with its over 200 conceptual models and over 150,000 attributes. In addition, it is an open research question how to best leverage detailed information on candidate matches to support discovery at the enterprise scale. Finally, integration in the large presents the following opportunity: Given an enterprise repository containing thousands of services, a large number of mappings among them, and many thesauri and ontologies, how can we use this growing knowledge base to make continual improvements in our ability to match and map? Recent work like [8] offers promising first steps toward such a vision.

5. Conclusions and Future Research

In this paper, we have presented a snapshot of an in-progress experimental effort between BEA Systems and MITRE to enhance the capabilities of a service-oriented data integration product (the BEA AquaLogic Data Services Platform) by guiding its interactive, XQuery-based data mapping capabilities using the composite matching recommendation and visualization features provided by a state-of-the-art, knowledge-based schema matching framework (Harmony). After examining some of the key challenges facing today's data architects and data integration engineers, we reviewed the capabilities and strengths of the two systems and identified the potential synergistic effects that we believe the combination of the systems – dubbed the AL$MONY system – can offer to users. We described the approach we are taking to combining these two systems and discussed some of the challenges that we face in doing so. We hope to complete this integration effort during the remainder of calendar year 2007 and to then evaluate its success by offering users of the current systems an opportunity to take AL$MONY for a "test drive" on some of their larger matching and mapping problems.

6. REFERENCES

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

[2] Borkar, V., Carey, M., Lychagin, D., Westman, T., Engovatov, D., and Onose, N. Query Processing in the AquaLogic Data Services Platform. *Proceedings of the VLDB Conference*, 2006.

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

[4] Carey, M. Data Delivery in a Service-Oriented World: The BEA AquaLogic Data Services Platform. *Proceedings of the ACM SIGMOD Conference*, 2006.

[5] Do, H. H. and Rahm, E. COMA - A System for Flexible Combination of Schema Matching Approaches. *Proceedings of the VLDB Conference*, 2002.

[6] Halevy, A. Why Your Data Won't Mix. *ACM Queue*, 3, 8, (October 2005), 50-58.

[7] Ilyas, I. F., Markl, V., Haas, P. J., Brown, P., and Aboulnaga, A. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. *Proceedings of the ACM SIGMOD Conference*, 2004.

[8] Madhavan, J., Bernstein, P., Doan, A., Halevy, A. Corpus-Based Schema Matching. *Proceedings of the IEEE International Conference on Data Engineering*, 2005.

[9] Mork, P., Rosenthal, A., Seligman, L. J., Korb, J., and Samuel, K. Integration Workbench: Integrating Schema Integration Tools. *Proceedings of the Second International Workshop on Database Interoperability (InterDB'06).* April 2006, Atlanta, GA.

[10] Mork, P., Seligman, L. J., Korb, J., Samuel, K., and Wolf, C. *Harmony Integration Workbench*. Submitted to US Patent Office, Ed., 2006.

[11] Rahm, E., and Bernstein, P. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10, 4, (December 2001), 334-350.

[12] Robertson, G., Czerwinski M., and Churchill J. Visualization of Mappings Between Schemas. *Proceedings of the ACM SIGCHI Conference.* April 2005, Portland, OR.

[13] Shipman, D. The Functional Data Model and the Data Language DAPLEX. *ACM Trans. Database Syst., 6*, 1 (March 1981), 140-173.

[14] www.bea.com/aler. *BEA AquaLogic Enterprise Repository*. BEA Systems Inc., 2007.