Galaxy: Encouraging Data Sharing Among Sources with Schema Variants

Peter Mork, Len Seligman, Arnon Rosenthal, Michael Morse, Chris Wolf, Jeff Hoyt, Ken Smith

The MITRE Corporation, USA

{pmork, seligman, arnie, mdmorse, cwolf, jchoyt, kps}@mitre.org

Abstract— **This demonstration presents Galaxy, a schema manager that facilitates easy and correct data sharing among autonomous but related, evolving data sources. Galaxy reduces heterogeneity by helping database developers identify, reuse, customize, and advertise related schema components. The central idea is that as schemata are customized, Galaxy maintains a derivation graph, and exploits it for data exchange, discovery, and multi-database query over the "galaxy" of related data sources. Using a set of schemata from the biomedical domain, we demonstrate how Galaxy facilitates schema and data sharing.**

I. INTRODUCTION

Communities often emerge in which participants collect and manage their own data using many variants of the same schema. For example, a health care economist may make data available in a spreadsheet or Access database. Others then adopt and customize the schema and populate it with their own data. These customizations may in turn be further customized by others.

To improve data sharing, many communities agree on a *core schema* but also allow individual systems and sub-communities to add their own information, for example using an XML wildcard (e.g., *xs:any*) or by adding a column to a spreadsheet template that has been downloaded from a website. The phenomenon exists at large scale as well: one schema intended for U.S. Air Force operations was copied and adapted to create databases for Army and Navy users with somewhat different needs. The National Information Exchange Model (niem.gov), a large collection of inter-related schemas (over 100 and growing) developed to encourage data sharing across diverse government agencies, is another example of large-scale schema reuse. This is a widespread design pattern in practice that we call *core and corona*, where a *corona* is an extension of the core schema that adds new entity types and/or properties of interest to a narrower community. The core schema provides a base level of interoperability across many systems, while participants still have the flexibility to respond rapidly to local data needs by creating coronae as needed.

The core/corona design pattern differs in significant ways from conventional data integration in which independently developed schemata are inputs to a post hoc schema matching process, and then coordinated using views or ETL scripts. Whereas there are many commercial tools and research results to support post hoc integration, there are few tools and little research to support the core/corona pattern. As a result, given two extensions of the same core, the standard approach for sharing data does not document the derivation and must match and map schemas from scratch.

Even worse, when systems within a single domain are not aided in sharing definitions, they are each likely to specialize a general concept (e.g., DrugTrial) incompatibly. For example, suppose two medical studies report drug reactions, but one includes all reports from doctors' visits while the other tracks only hospitalizations. There is no way to compare or combine their results—one needs to change which data are collected.

Thus, the main point of our research is to increase the likelihood that multiple systems will collect and interpret data in compatible ways. By doing so, we make it more likely that data from these systems can be compared meaningfully, and we make it easier for these systems to share data. In other words, we seek to reduce both fundamental incompatibilities and the need for expensive post hoc integration.

For example, consider the scenario depicted in Figure 1 in which node *A* represents the core schema and the remaining nodes represent coronal extensions. Systems that directly implement the core can share their data with any member of the community with no additional work needed. Similarly, participants that use schema *B* can share their data upstream by removing the insurance attribute and downstream by removing the address attribute. Because the various participants use the same representations for the remaining attributes (such as language), we do not need to deal with value conversions.

Knowledge of schema derivations can also ease sharing across sibling nodes. For example, given a derivation graph

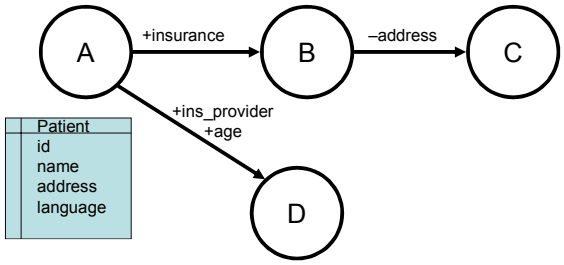

Figure 1: Sample sharing scenario in which the core schema (A) is extended in different ways to generate schemata B and D. Schema B is further revised to generate schema C.

such as Figure 1, tools could fully automate some data sharing from *B* to *D* (i.e., all information defined in *A*), because the matches are already known with complete certainty. Additional sharing would require some traditional data integration (e.g., to determine that insurance and ins_provider capture the same information), but we have greatly reduced the size of the match/map problem by restricting it to the coronae. In addition, matching is simplified because so much of the attributes' context is already known to match. (Since traditional matching is not our main contribution, it will not be included in the demo). In other words, using the core and corona design pattern, we are able to realize significant data interoperability with little to no integration effort.

However, despite the potential long-term interoperability benefits of tracking schema reuse, developers are unlikely to generate the necessary inter-schema relationships manually. Thus, we demonstrate development tools that a) encourage schema developers to reuse schema specifications whenever possible and b) automatically capture the necessary derivation metadata.

This demonstration presents Galaxy, a collection of tools based on these desiderata. Using Galaxy a schema developer can a) search and browse for relevant schema specifications and b) both make local customizations as well as publish those customizations back to the community.

Thus, Galaxy is a community schema manager (currently being deployed to support MITRE's customers) that facilitates easy and correct data sharing among autonomous, but related, evolving data sources. It accomplishes this by:

- reducing heterogeneity by helping database developers identify, reuse, customize, and advertise related schema components, and
- explicitly managing the derivations among families of related schemata, which enables
- using the knowledge of inter-schema relationships to support multi-database query and data exchange over all related data sources that can answer the query.

Galaxy directly supports the core and corona design pattern. It is *not* intended as a component of (or replacement for) a schema matcher—it is a separate system that greatly reduces the need to perform post hoc integration across highly heterogeneous specifications. Ultimately, we envision tool suites that support both core and corona and traditional data integration. In fact, we are working toward this vision: Galaxy is one component of *Open II*, an open source information integration toolkit under construction by MITRE, Google, and other collaborators.

In this demonstration proposal, we first describe the Galaxy models for tracking derivations and querying across schemata. We then provide details of the conference demonstration. Finally, we describe related work.

II. Galaxy Repository Model

Galaxy's repository tracks schema derivations and deviations. It stores 1) a directed acyclic graph in which nodes are schemata and arcs indicate derivation (described below), 2) a 1-to-many relationship between a schema and the data sources that instantiate the schema and 3) metadata about data sources (e.g., how they can be queried). Figure 2 illustrates a derivation graph as shown by the Galaxy user interface. A rectangle with a blue S is a schema, an arrow indicates derivation, and a blue rectangle represents a data source that instantiates the schema to which it is linked. For example, two of the schemata in Figure 2 are instantiated: the Organism schema and the Protocol Subject schema. The Human Patient, Chemlab Subject, and Imaging Subject schemata are not instantiated.

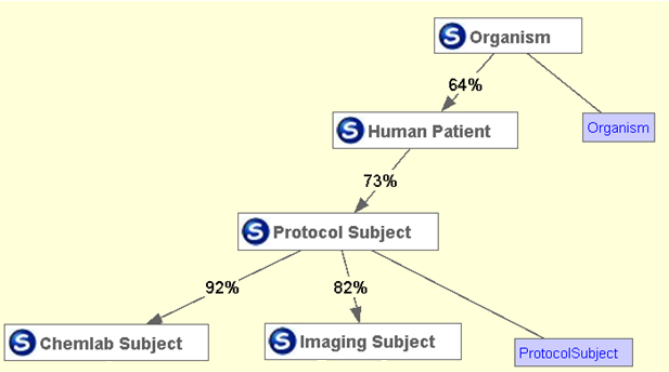

Figure 2: Galaxy Schema Extension Graph, showing a navigation interface for extensions and derivations of the Organism core, with data sources shown in blue. Derivation edges between schemata are labelled with the percentage of the components of the derived schema that are inherited from the source schema

A schema consists of two kinds of constructs: structural constructs (attributes, relationships, and domain values in our demo) and constraints (e.g., primary and foreign keys). A *component* consists of a structural construct and its associated constraints. Structural constructs have an associated semantics that may be represented as text or more formally as a reference to an element of an ontology. A corona can extend multiple cores, thereby establishing a derivation DAG. The relationship between the Galaxy repository metamodel and other popular metadata models is shown in Table 1.

A coronal schema is obtained by *importing* and *editing*. Each imported component retains its semantics and identity (e.g., using a URI). By maintaining identity, Galaxy greatly reduces the need for post hoc integration. A component can subsequently be edited by adding or removing additional sub-structures (such as attributes) or by adding constraints. In effect, the editing process identifies a view component that is being shared. When importing components rather than an entire schema, the system ensures that the result is a well-formed schema. As shown in Figure 2, we also display for each corona the percentage of its components derived from its respective cores using the formula (# of core structural components)/(# of core and coronal structural components). For example, 92% of the components in the Chemlab Subject schema are also found in the Protocol Subject schema.

To further illustrate, consider Figure 1: Schema *B* is derived from *A* and imports the Patient entity with attributes id, name, address and language along with their semantics and any

associated constraints. *B* customizes *A* by also adding the attribute insurance. *C* imports everything in *B*, except that it removes address. Note that while both *B* and *D* add information about insurance, these are distinct components that may have different semanticsrepresentations. For example, *B*.insurance could have a {Yes, No} domain, while *D*.ins_provider could be {Aetna, BlueCross, …}.

The Galaxy model allows one to associate additional constraints with data sources. For example, one might have a schema PatientInfo with attributes Person.record# and Person.zip. PatientInfo may contain a constraint that Person.record# is not null. An associated data source VirginiaPatients may have an additional constraint that Person.zip must be a valid zipcode in the state of Virginia.

III. Query Model and Implementation

We first describe how queries are posed, and what they mean. We then describe how queries are forwarded to relevant data sources, and finally, describe the handling of overloaded domain values.

The Galaxy query model allows users to choose a schema against which to pose their queries, and to select a subset of the data sources as *candidates*. In practice, users usually select {all available data sources} or "local data only".

Informally, the query semantics are to *ascertain* from each candidate source which tuples satisfy the query predicate, and return values for any of the requested attributes that are available for each tuple (and to fill others with nulls). *Can be ascertained* denotes best efforts, but not completeness. In effect, the instance set associated with a schema consists of the set of all instances semantically compatible with that schema, from any candidate data source. The language currently allows queries that conform to a single source schema, with conjunction, disjunction, and negation of predicates (cross-schema joins seem a straightforward extension).

Given a specific query, the next task is to determine which data sources can sensibly contribute data to the answer. We assume sources are vanilla SQL databases, not modified to be aware of the Galaxy model. We use the term *item* to refer to any entity, attribute, or domain value that appears in the query text. For each candidate data source, Galaxy determines which of the query's items the source understands and constructs a query that it can interpret, whose results are guaranteed to satisfy the original query. If Galaxy cannot construct such a query, the source is ignored.

As an optimization, we determine *query sensibility* (i.e., it is worth evaluating by a particular source) using the following recursive definition. A query is sensible if its predicate is sensible. For an atomic predicate to be sensible, every item mentioned must be present in the data source, or else the query must include IS NULL tests. A conjuctive (respectively, disjunctive) predicate is sensible if all (at least one) of its terms is sensible.

Consider Figure 1 and a query for which address is a mandatory construct. It is sensible to forward that query to any instances of schemata *A*, *B*, and *D*. It is not sensible to forward the query to instances of schema *C* because address has been removed by *C*.

We provide a careful treatment of domain values in queries against autonomous sources. Each domain value has a unique Galaxy identifier (obtained by prepending the schema name where that value is defined). Galaxy can thus detect that local values from unrelated extensions to a domain represent different values. For example, suppose *A*'s language domain is {En, Ch} which all schemas derived from *A* understand to be English and Mandarin Chinese. Then, *B* and *D* each add "Sw", *B* meaning Swedish and *D* meaning Swahili. Now a query predicate *(language=Ch)* is disambiguated to A.Language.Ch, which is also sensible to forward to *D*. On the other hand if B poses a query with a term: *(language= "Sw"),* Galaxy determines that B.Language.Sw is not in the Domain(*D*); therefore, the query is not sensible to D.

To make such queries sensible, the system has ways to rewrite them to exploit the intersection of the domains. For example, a predicate P (defined in schema B) might be rewritten for schema D by adding a (redundant) disjunct P_D that *is* interpretable in D but implied by P. For example, if predicate P were "language name starts in A-F", P_D would be *language.Name is in (Domain(B) ∩ Domain(D)) and name starts in "A-F"*. (A slightly different rewrite helps with negated queries.) Now both En and Ch would be returned. On its own data, D need not test membership in Domain(D)

IV. Demonstration

The proposed demonstration will show how database designers and end users benefit from Galaxy through four main processes: *discovering* potentially relevant schemata, *navigating* the derivation graph to identify a schema to reuse, *extending* that schema to accommodate application specifics (and documenting the result), and finally *querying* instances of that schema for data. Details of each process follow:

Table 1: Relationship between Galaxy repository metamodel and other common metamodels

Galaxy	SQL	XML	RDF	UML
Entity	Table	Complex type	Class	Class
Attribute	Column	Attribute	Datatype Property	Attribute
Relationship	Inferred from FK/PK	n/a	Object Property	Association
Subtype	n/a	Extends	Subclass	Subclass
Containment	n/a	Element	n/a	Composition
Domain	Built-in types	Simple type	Built-in types	Atomic types
Derivation	n/a	n/a	n/a	n/a

Discovery: To identify a core set of schema elements to extend to create a new source, a designer enters pertinent keywords. Galaxy returns schemata and extensions to these schemata which match the entered text, including annotations about schema entries, attributes, and domain values.

Navigation: Once one or more relevant schema families have been identified, the designer can explore the schema graph to identify the particular schema that is most appropriate to serve as the core for their specific application, as shown in Figure 2. Galaxy users can browse the derivation graph, inspect particular schemata, and compare them using a graphical diff.

Extension: A designer creates a new schema by importing the entities and attributes of one or more existing schemata in a family. This new schema is now a descendant of each of the schemata whose entities and attributes it imported. The new schema can be customized by adding or subtracting elements in a graphical environment provided by Galaxy. Galaxy understands where two schemata employ the same component.

Query: A schema, as populated by a selected set of sources, can be queried. The query system presents an interface from which any schema can be queried. An inference engine identifies those data sources to which a query can sensibly be forwarded. These databases receive the query and forward all results back to the query system, which combines the results before presenting them to the user.

The demonstration uses a family of biomedical schemata, whose core includes patient identifying elements. We derive coronal schemata by importing the core and adding elements necessary for different types of patient studies. For example, to create a derived schema for a neuro-imaging study, the designer adds schema elements pertaining to image modalities. Numerous health and medical study schemata extend from the core schema, and many of these schemata have data sources associated with them. We then illustrate end users performing a variety of queries. Galaxy's distributed query facilities bind queries to appropriate sources and combine the results.

V. Related Work

By creating schemata with known matches, we reduce the need for post hoc integration. We thus work side by side with tools based on the enormous literature on schema matching and mapping (e.g., [1,3,5,6,9]). We show the value, for non-IT specialists, of a *small* set of schema extension and edit operations; in contrast, model management [2] provides a rich, general purpose, very complex framework [8]. Galaxy introduces a simple extension operator for which the mapping across versions is explicit.

Galaxy is related to schema versioning [10] and evolution [3]: whereas in Galaxy each schema is immutable, each derived schema can be viewed as a new version of its parent. In addition, prior schema versioning research aims to ease migration of instance-sets to more current versions. In contrast, our work versions only schemata, using them to support data sharing over autonomous but related sources. Also, in contrast to prior work, we provide explicit support for discovery, reuse, and extension of existing schema components.

Google Base [7] offers flexible sharing and query over single entity types with an extensible, untyped set of properties. It provides very forgiving, simple structured query over data that might otherwise support nothing better than text search. In contrast, Galaxy supports the needs of communities that have somewhat greater semantic cohesiveness and need more powerful query, while still allowing participants to extend existing schemata to meet their unique requirements.

VI. Summary and Future Research

Galaxy demonstrates reuse and customization of components in a family of related schemata. It uses knowledge of schema variants to provide easy and correct data sharing among autonomous but related, evolving data sources. We have demonstrated Galaxy to several customers; their feedback convinces us that it addresses a real and important problem.

Future plans include releasing Galaxy as a component of *Open II*, an open source information integration toolkit currently under construction. A key research issue is how to visualize the results of schema search. For example, consider a query for a few concepts of interest over a repository of thousands of schemas with hundreds of schema derivation graphs. We are creating algorithms to maximize cohesiveness while minimizing redundancy. We aim to make the results intelligible to the user so that she can quickly identify the most promising components to import into a new schema under construction.

References

[1] Aumueller, D., Do, H., and Rahm, E. Schema and Ontology Matching with COMA++. In *SIGMOD*, 2005.

[2] Bernstein, P. Applying Model Management to Classical Meta Data Problems. In *CIDR,* 2003.

[3] Dhamankar, R., Lee, Y., Doan, A., Halevy, A., Domingos, P. iMAP: Discovering complex semantic matches between database schemas, In *SIGMOD*, 2004.

[4] Franconi, F., Grandi, F., and Mandreoli, F. A Semantic Approach for Schema Evolution and Versioning in Object-Oriented Databases. *Computational Logic*, 2000.

[5] Hernández, M., Popa, L., Velegrakis, Y., Miller, R., Naumann, F., Ho, C.-T. Mapping XML and Relational Schemas with Clio. In *ICDE*, 2002.

[6] Madhavan, J., and A. Halevy. Composing Mappings Among Data Sources. In *VLDB*, pages 572-583, 2003.

[7] Madhavan, J., Shawn, J., Cohen, S., Dong, X., Ko, D., Yu, C., Halevy, A. Web-scale Data Integration: You can only afford to Pay As You Go. In *CIDR*, 2007.

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

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

[10] Roddick, J. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology.,* 37(7), 1995.