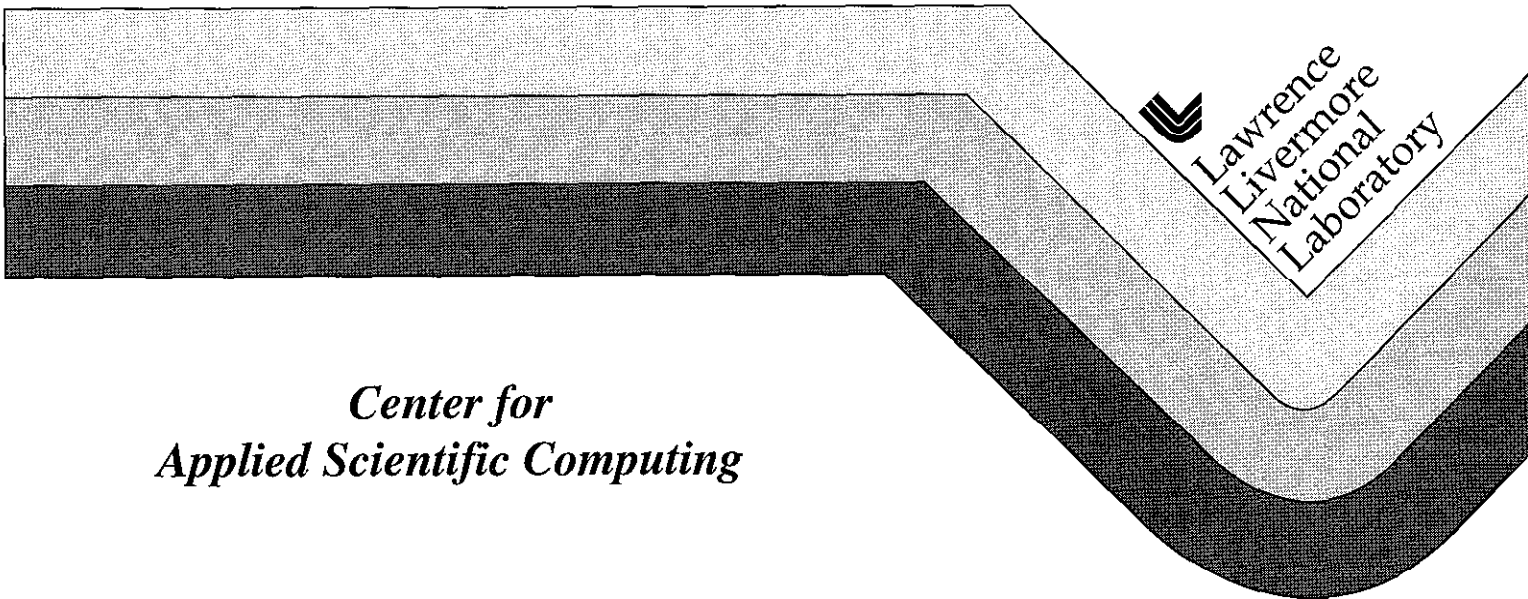


# Metadata Based Mediator Generation

T. Critchlow  
M. Ganesh  
R. Musick



*Center for  
Applied Scientific Computing*

#### DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

#### PREPRINT

This is a preprint of a paper that was submitted to the Third International Foundation on Cooperative Information Systems Conference on Cooperative Information Systems, New York, NY, August 20-22, 1998. This preprint is made available with the understanding that it will not be cited or reproduced without the permission of the authors.

## Metadata Based Mediator Generation

T. Critchlow, M. Ganesh, R. Musick  
*Center for Applied Scientific Computing*  
*Lawrence Livermore National Laboratory (LLNL)*  
 {critchlow | ganesh | rmusick}@llnl.gov

### Abstract

Mediators are a critical component of any data warehouse, particularly one utilizing partially materialized views; they transform data from its source format to the warehouse representation while resolving semantic and syntactic conflicts. The close relationship between mediators and databases, requires a mediator to be updated whenever an associated schema is modified. This maintenance may be a significant undertaking if a warehouse integrates several dynamic data sources. However, failure to quickly perform these updates significantly reduces the reliability of the warehouse because queries do not have access to the most current data. This may result in incorrect or misleading responses, and reduce user confidence in the warehouse. This paper describes a metadata framework, and associated software, designed to automate a significant portion of the mediator generation task and thereby reduce the effort involved in adapting to schema changes. By allowing the DBA to concentrate on identifying the modifications at a high level, instead of reprogramming the mediator, turnaround time is reduced and warehouse reliability is improved.

**Keywords:** mediators, metadata, data warehouse, scientific databases, ontologies, materialized views.

### 1. Introduction

One of the most formidable problems faced in accessing data from multiple heterogeneous sources is resolving schema and data conflicts. In evolving scientific domains such as genetics, this problem is compounded by frequent source schema changes frequently. The DataFoundry project at LLNL is aimed at providing acceptable performance under these demanding situations. DataFoundry uses a mediated data warehouse architecture, supported by a carefully designed domain-specific ontology. As a result, it is able to rapidly adapt to source schema changes by automatically generating mediators.

Mediators are critical components of data warehouses. They are responsible for transferring data

from the source databases<sup>1</sup> to the warehouse, and for resolving all conflicts between the source and target representations. In traditional data warehouses, mediators regularly repopulate the warehouse and ensure that the warehouse data remains up to date. In a warehouse using partially materialized views, however, the mediators are also responsible for dynamically providing access to non-materialize data. This additional responsibility makes high reliability imperative since failures directly affect the usability of the warehouse. Unfortunately, whenever a schema changes, the associated mediators need to be updated to reflect the modifications. Until these changes are incorporated warehouse usability is compromised; in the best case, queries return incomplete or slightly out of date data; in the worst, misleading or incorrect results. It is critical for the long term feasibility of the warehouse to ensure these interruptions are as short as possible, and do not adversely affect the perceived reliability. In domains where schema changes are infrequent this is not a significant concern. However, in highly dynamic scientific domains frequent schema modifications are a reality that must be faced.

To evaluate the effects of different design decisions, DataFoundry is developing a prototype data warehouse to aid in genetics research. Genetics is an ideal domain in which to pursue our research for two reasons. First, it is an evolving scientific domain in which the interactions of the underlying data are not yet fully understood. As experimental techniques are developed, and understanding of the data grows, the database schemata adapt to reflect this new knowledge. Given the speed of discovery in this area, the corresponding rates of schema change are extremely high: based upon previous efforts, we anticipate one schema modification every 2-4 weeks once all of the desired sources are integrated. Second, by successfully providing a warehouse linking several existing community databases, DataFoundry will provide an invaluable resource to genetics researchers. While this is not critical to performing the necessary computer science research, it provides additional motivation to ensure the approach is practical.

<sup>1</sup> We use *database* to refer to any managed collection of data including, but not limited to, flat files, relational databases and object-oriented databases.

Due to the frequency of schema modifications in this domain, maintaining the mediators using straightforward techniques results in the warehouse having only partial access to information for an unacceptable amount of time. DataFoundry makes extensive use of a carefully designed ontology and overcomes this problem by automatically generating the mediators directly from the metadata. Thus, when a schema changes, the DBA needs to update only the ontology, as compared to directly modifying the mediator code. This has the additional benefits of improving code reuse, providing a consistent API to wrappers, and providing a useful knowledge base for other applications such as a high level interface to the warehouse and automatic schema evolution.

This paper describes the DataFoundry metadata representation and how it is used to automatically generate mediators, thereby reducing the effect of source changes and improving access to heterogeneous data sources. A comparison with other research efforts is provided next in section 2, followed by a brief overview of the DataFoundry architecture. Section 4 describes the information represented in the ontology, and Section 5 outlines how it is used to generate the mediators. Finally, we conclude with a summary of our approach and outline future research directions.

## 2. Related efforts

We have selected a mediated data warehouse approach with partially materialized views as the DataFoundry architecture. Our goal is to relieve the users of the responsibility for data integration and data format mismatch issues and present a coherent view of the underlying data. To automate the process of mediator generation DataFoundry makes extensive use of an ontology framework. The remainder of this section highlights a few of the many research projects in these areas and, where appropriate, compares them to DataFoundry.

Mediators [20] are software agents which act as translators for data encapsulating all the routine work of converting data from one format to another. While, in theory, these conversions may be arbitrarily complex, in practice they are often limited to trivial operations. Mediators may also include the ability to identify the data sources which provides the requested information and dynamically forward the request to them.

The TSIMMIS [3][6][7] project at Stanford uses mediators for transformation of data from several diverse sources. TSIMMIS, like most mediated architectures (including InfoSleuth [2], Disco [13] and DIOM [19]), does not provide a global schema and delegates conflict resolution to the end user. A serious problem with pure mediated architectures is data source failure; when a source is unavailable, incorrect query results may be returned. [19] attempts to address this problem by

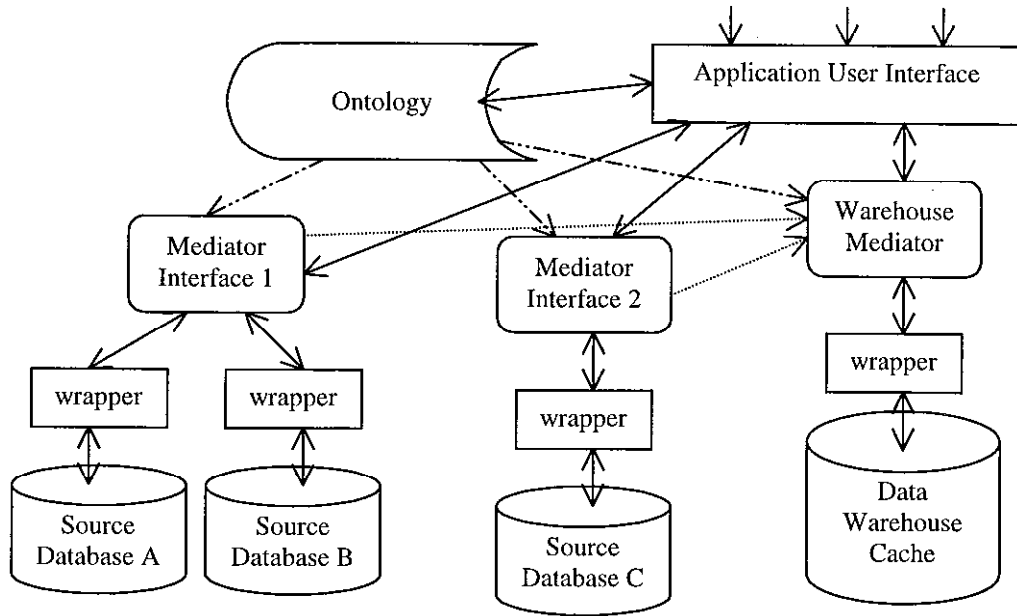
returning the uncompleted portion of the query, which can be reevaluated later. DataFoundry takes a different perspective. A global schema is provided on the assumption that the end user will not be familiar enough with the individual sources to resolve subtle conflicts. Further, by utilizing the warehouse as a local cache, the effects of an unavailable source can be significantly reduced.

Ontologies [8][9] store knowledge about real-world objects and their relationships. They enable high-level queries to be posed directly against a database, instead of embedding them in application programs. Cyc [12] is one of the first, and best known, ontology-based projects. It created a large base of common-sense knowledge that works reasonably well in many environments. Unfortunately, it lacks the specialized vocabulary required to be effective in terminology-rich domains. When ontologies are used in specific domains, such as the medical field, the challenge is to conceptually link multiple information resources that use different terminology [5]. The OBSERVER project [14][15] is aimed at providing a framework for interaction among existing ontologies in a global information infrastructure. This project is aimed at bibliographical information and uses a thesaurus to resolve terminological differences among the ontologies. DataFoundry links biological databases that do not provide significant ontological information [5], and implements a global ontology as a facilitator for information integration from disparate sources. Other applications of ontologies have been in linguistics-related fields to help natural language processing [16]. While DataFoundry intends to use the ontology not only as a resource for generation of mediators, but also to support the query processor and guide schema evolution, applications such as NLP are not currently being considered.

Materialized views [10][18] of source data have long been used as a mechanism for fast access to data. To maintain consistency a well-defined view update policy, based on the number and importance of changes to the source, is required. In data warehouses, partially materialized views [1] have been proposed as a method to reduce data communication between the sources and the warehouse. DataFoundry uses partially materialized views to improve the query response time to the most frequently accessed data. Mechanisms to dynamically refresh warehouse data when it is not available or is inconsistent are also included.

## 3. The DataFoundry architecture

The goal of DataFoundry is to provide integrated access to multiple, evolving, domain databases through an integrated interface. To facilitate this, we have chosen an architecture which combines the advantages of federated databases [17] as well as those of data warehouses [11].



**Figure 1 DataFoundry Architecture**

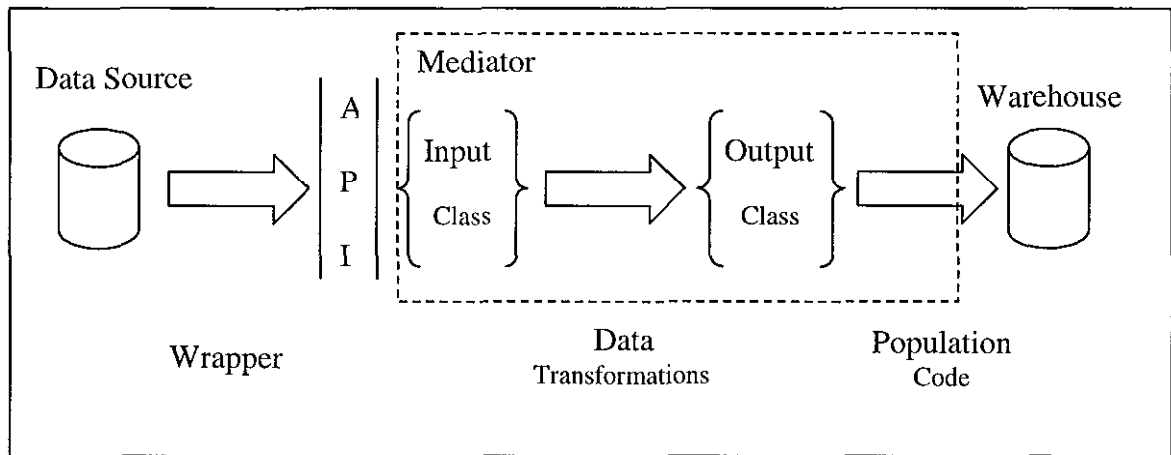
Federated databases provide a global schema for the underlying source databases, each of which retain control and management of their data. Queries posed against the global schema are translated into individual queries against the source databases, and results obtained from these are combined together before being returned to the user. This query mechanism is made possible by the mappings between the information contained in the source databases, maintained in the global query processor. Traditional data warehouses, on the other hand, materialize the global data in a local cache which permits fast access to the warehouse data. Data from different sources is merged together in a batch operation and stored at the warehouse to provide immediate responses to queries. This scheme requires frequent refreshes to the local cache if the source data changes often.

DataFoundry seeks to support scientists in evolving research areas where the source data and schemata change frequently, that neither a federated architecture nor a conventional data warehouse are suitable information architectures. To quickly adapt to the changes in source database schemas DataFoundry uses a mediated [20] data warehouse architecture supported by a domain ontology. In this architecture, only data that is frequently accessed is materialized in the warehouse cache. This provides fast access for most queries, while maintaining the consistency of data with the source databases. The overall dataflow architecture in the Data foundry is shown in Figure 1. The main components in this architecture are the ontology, the mediator interface to the source databases, and the application user interface and data warehouse.

The ontology is a description of the information contained in the data sources and their relationships. While the ontology is external to the data flow, it is used to generate the mediators and provides a high-level interface to the data. The details of the ontology are presented in the remaining sections of this paper.

To access data from the warehouse, an application queries the application user interface. The interface consults the ontology to determine whether the data is available in the warehouse or if it needs to be dynamically retrieved from the source databases. Access to data sources is done through the mediator interfaces which transform the data from the source format to the DataFoundry format and return the results to the warehouse.

Figure 2 outlines the steps involved in loading the warehouse: parsing the data, transforming it to the warehouse format, and entering it into the warehouse. In practice, these steps are not always distinct. Often, a single program will parse the input file, and transform the data before storing it in an internal specification. This internal representation can then be entered into the warehouse after any transformations to the storage format if required. Intermingling of wrapper and mediator is permitted because the mediator API is rarely defined. However, a carefully designed API is critical to reduce the maintenance requirements of the warehouse; it allows the ontology and warehouse to evolve without affecting the wrapper. DataFoundry uses a well-defined API, based on the ontology concepts, to provide a clear separation between the mediator and wrapper functions. DataFoundry uses an object-oriented model for the description of data items internally, without placing any



**Figure 2 The Integration Process**

restrictions on the data model used for data storage in the warehouse or in the source databases. The wrappers are responsible for the translation between the underlying data model and the global object model.

Mediators in the DataFoundry are responsible for transferring query requests to appropriate data sources and managing the integration of information returned from the different sites. In addition, they are also designed to act as managers for detecting changes in source databases and propagating updates in the materialized data to the warehouse cache. The mediators also update the warehouse metadata to maintain data consistency in DataFoundry.

#### 4. The ontology

The DataFoundry ontology is a collection of Ontolingua [8] classes and instances that define three types of knowledge: generic descriptions of databases, concepts and associations; specific instances of these descriptions; and domain-specific abstractions representing knowledge about a particular field. The generic descriptions are provided for completeness, and are not discussed further. Instead, we focus on four distinct, but inter-related concepts that provide the metadata used to generate the mediators: domain-specific abstractions, database descriptions, mappings between the abstractions and descriptions, and transformations between different abstraction representations. As databases evolve and additional data sources are integrated, new database descriptions and mappings are defined. These may, in turn, require specifying new abstractions, updating the attributes associated with an existing abstraction, and defining new translation methods. The remainder of this section describes these components in detail, using the examples shown in Figure 3 and Figure 4, while Section 5 describes how they are used to generate the mediators.

##### 4.1. Domain-specific abstractions

Abstractions are the core of the domain specific knowledge represented by the ontology. Conceptually, an abstraction encapsulates the different components and views of a particular domain-specific concept. Practically, an abstraction is the aggregation of all of a concept's associated attributes and representations, as presented in the participating databases. As such, the abstractions contain a superset of the information contained in any individual database.

Each abstraction is an Ontolingua class that inherits, directly or indirectly, from a distinguished abstraction class. The abstraction's attributes are grouped into characteristics that combine related attributes and alternative representations of the same attribute. The genome abstraction shown in Figure 3 describes the characteristics and attributes associated with the *atoms* abstraction within the *genomics* domain. Notice that while the *id*, *flexibility*, *element* and *alternative\_position* characteristics have only one attribute associated with them, the *position* characteristic has three, which combined represent a position in 3-D space using Cartesian coordinates. If there were multiple representations of the same characteristic (e.g. a *long* element name) there would also be multiple attributes in the same characteristic. While this grouping has no affect on the mediator, it provides a mechanism to document the conceptual relationship between these attributes.

This example also highlights two interesting features of the attribute representation. First, it demonstrates how to define attributes of complex data types, encouraging a natural description of the domain specific concepts. Consider the *alternatives* attribute; instead of a primitive data type (i.e. integer, character, string, float, double), it is defined as a data structure representing the Cartesian coordinates and flexibility of the alternative position, as well as the probability of the

|   |   |   |
|---|---|---|
| <pre>(define-instance dw (relational-db) :df (= dw '   ((atom     "The atoms table describes     the actual position of each     atom in the sequence"     ((self oid key)      (model_res oid       (res_in_model self))      (x float 1)      (y float 1)      (z float 1)      (temp float 0)      (element (string 4) 1)))     .....))</pre> <p style="text-align: center;"><b>Warehouse Descr.</b></p> | <pre>(define-instance map (translation) :df (= map '(   ((genomics atoms)    (dw atom res_in_model alts)   ((atoms res_key)    (res_in_model residue))   ((atoms mod_key)    (res_in_model model))   ((atoms short_el)    (atom element))   ((atoms x) (atom x))   ((atoms y) (atom y))   ((atoms z) (atom z))   ((atoms temp)    (atom temp)))   .....))</pre> <p style="text-align: center;"><b>Mapping</b></p> | <pre>(define-instance gene-abs   (genomics-details) :df (= gene-abs '(   genomics   ((atoms     (id (warehouse_key oid))     (position (x float)               (y float)               (z float))     (flexibility (temp float))     (element (short (string 4))))   (alternative_position     (alternatives ((x float)                   (y float)                   (z float)                   (temperature float)                   (probability float)) N)))   .....))</pre> <p style="text-align: center;"><b>Genome Abstractions</b></p> |
|---|---|---|

**Figure 3 Examples of Ontology Data**

atom being there. Second, each attribute has an arity associated with it, representing the number of values it can or must have. The possible values are:

- **key**: the attribute is single values, required and unique
- **(class attr)**: the attribute is single valued and optional, but if it exists, it must also occur in the *attr* member of *class* (i.e. it is a foreign key)
- **0**: (the default) the attribute is optional and single valued
- **#**: the attribute has exactly the number of values specified by the #
- **N**: the attribute is optional and multi-valued
- **1\_N**: the attribute is multi-valued but must have at least 1 associated value

To ensure that abstractions remain a superset of the component databases, incorporating a new database requires updating abstractions in two ways. First, any previously unknown concepts represented by the new data source must be incorporated into the class hierarchy. Second, any new representations or components of an existing abstraction must be added to its attribute list.

#### 4.2. Database descriptions

Database description are language independent definitions of the information contained within a single database. These definitions are used to identify the translations that must be performed when transferring data between a specific data source and target. Eventually, they will also be used as hints for automatically creating a new database description after a schema modification, similar to those used by [4].

As the warehouse description in Figure 3 shows, the ontology representation of a database closely mirrors the physical layout of a relational database. In this example, the class (table) name, *atom*, is followed by a comment and a list of associated attributes. There are two advantages to using this independent representation of the data. First, the database attributes have the same functional expressibility as the abstraction attributes described above. As a result, they are able to represent non-relational data sources, including object-oriented databases and flat files; a crucial capability when dealing with a heterogeneous environment. Second, the ability to comment the database descriptions improves warehouse maintainability by reducing the potential for future confusion. Class comments may be used to clarify the interactions with other classes, define or refine the concept associated with a table, etc.. These comments are complimented by attribute comments which, while infrequently used for abstraction attributes, provide additional metadata about the attribute's purpose and representation.

As databases are integrated into the warehouse, their descriptions must be entered into the ontology. Furthermore, as their schemata change the ontology must adapt appropriately. Currently, this modification is made by the DBA, but we plan to investigate automating this process. Because of the similarity between the ontology and relational formats, it is possible to automatically generate most of the ontology description directly from the metadata associated with most commercial DBMSs; obviously the DBA must still explicitly enter any comments they wish to provide. Unfortunately, because

```

(define-instance genome-transformations (abstraction-enhancement)
  :def (= genome-transformations
        ("/home/critchlo/data-warehouse/ontology/lib/genome.lib"
         (amino_acid
          (translation-methods (full_to_one_char)
                               (full_to_three_char)
                               (one_char_to_full)
                               (three_char_to_full))
          (class-methods (three_char_to_one_char
                          ("one_char" character)))
          (class-data ((name_conversion_table
                        ("one_char" character)
                        ("three_char" (string 3))
                        ("full_name" (string 40))) 28)
                       ({ {"A", "ALA", "Alanine"}, {"R", "ARG", "Arginine"},
                          {"N", "ASN", "Asparagine"}, {"D", "ASP", "Aspartic acid"},
                          ...}))....)))

```

**Figure 4 Transformation Definitions**

most flat file databases do not maintain any metadata, the ontology description must be manually defined.

can be overcome by careful examination of the attributes and use of multiple SQL queries.

### 4.3. Mappings

Mappings identify the correspondence between database descriptions and abstractions at both the class and attribute levels. Because abstractions are a superset of the individual databases, there is always a direct mapping between database and abstraction attributes. Due to representational differences, however, an abstraction may be split across several database classes and a single database class may be related to several abstractions.

Figure 3 demonstrates how the warehouse *atom*, *res\_in\_model*, and *alts* database classes are mapped to the *atoms* abstraction. By default, the classes are joined on the key / foreign relationships identified in the database description. Ambiguity about which attributes should participate in the join is resolved by explicitly defining the join conditions. Thus, the implicit mapping in Figure 3 could be replaced with:

```

((= (atom model_res) (res_in_model self))
 (= (atom self) (alts atom)))

```

A similar notation permits dereferencing complex attributes of an abstraction to map to all of the attributes in a database class.

While converting single table mappings to SQL is easy, care must be taken when converting multi-table mappings with optional attributes. Simply joining the participating tables and using the results identifies only a subset of the desired instances, because those that do not have the optional attributes will not be identified. For example, *atoms* that do not have *alternative\_positions* would be lost in the join of *atom* and *alts*, even though they are valid instances of the abstraction. This problem

### 4.4. Transformations and other user extensions

Transformations describe which attributes contain the same data, but in different formats, and identify the methods that can be used to translate between them. The ontology does not define these methods explicitly, but rather just records their names. DataFoundry uses a naming convention to identify the attributes manipulated with a particular method. An alternative, more verbose, approach would be to explicitly associate the participating attributes with each method. In either case, these methods are restricted to operating only on class member variables and, as such, do not require any parameters. To provide the maximum flexibility, DataFoundry allows two types of other extensions to be associated with an abstraction: class methods and class data. This information is shared by all instances of an abstraction.

Figure 4 presents the extensions for the *amino\_acid* abstraction. The actual method definitions are located in the *genome* library. A simple naming convention of *source\_attribute\_name\_to\_target\_attribute* permits the attributes associated with each method to be easily identified. It is important to note that a sequence of method invocations may be required to obtain the desired representation. For example, the conversion from *three\_char* to *one\_char* takes two steps; first converting from *three\_char* to *full*, then from *full* to *one\_char*. The class method *three\_char\_to\_one\_char* returns the corresponding *one\_char* value for a given *three\_char* value; however, since it can be invoked



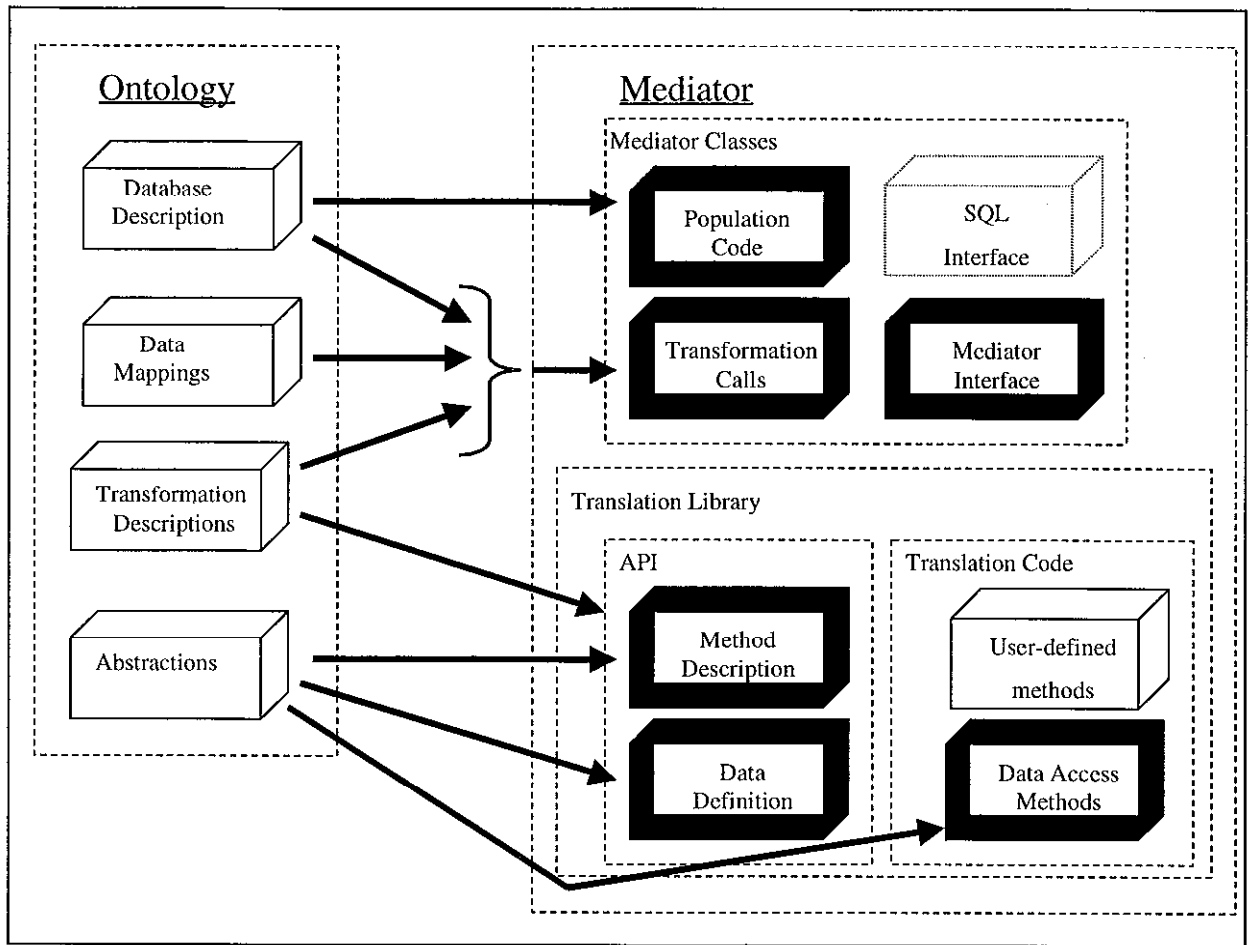


Figure 5 The Ontology and Mediator

without an associated *amino\_acid* instance, it may not modify the instance attributes as the sequence of transformation method invocations would. This method would be used in another class, such as *chain*, that requires this functionality, but does not require an instance of this class. Class data is useful for providing information such as a translation table that does not vary between instances of the class.

There are two benefits to identifying these methods in the ontology. First, and most obvious, it provides the ontology with the final piece of knowledge required to generate the mediators. However, a subtler benefit is the combination of the abstraction methods into a single library. By explicitly identifying these methods, and defining them in a single location, code re-use is encouraged and maintenance costs reduced.

## 5. Automatic mediator generation

Once the ontology has been defined, an ontology engine (OE) is used to generate the C++ classes and methods that comprise the mediator. Figure 5 outlines how the ontology concepts discussed in the previous

section relate to various components of the mediator. As shown, the mediator functionality is decomposed into a translation library and a set of mediator classes. The translation library represents the classes and methods associated with the ontology abstractions, while the mediator classes are responsible for performing the data transformations. The API available to the wrapper is a combination of the mediator class and translation library APIs. The process of obtaining these components from the ontology is relatively straightforward, and is therefore only discussed briefly below.

The translation library encapsulates the class definitions and methods associated with the domain-specific abstractions. The OE defines a distinguished *abstraction* class, and one class for each ontology abstraction. The inheritance hierarchy is the same as the ontology abstraction hierarchy, except that the base classes inherit from *abstraction*. This provides all classes with a minimal amount of functionality, including access to both the source and target databases. The data members associated with a class correspond to the abstraction attributes; static data members are used to

represent the class-data extensions. Abstractions used as multi-valued attributes have an additional data member, *next\_ptr*, which references the next element in the list. Structures are defined for complex data types, and are named based on the corresponding attribute name. The OE also defines two data access methods for each attribute: one to read it, the other to write it. The class definition includes public definitions of these methods and static declarations for the class-method extensions.

The mediator classes are only slightly more difficult to generate. For each defined source – warehouse pair, a mediator class is generated to perform the data transformations and enter the data into the warehouse. Different classes are required because the transformations vary depending on the source format, and using a pure data-driven approach to dynamically identify the appropriate transformations would be too slow. For each class, a method is defined to take each of the top level abstractions and convert them to the warehouse format.

The set of required transformations is obtained by comparing the attributes provided by the data source to the ones required by the warehouse. If a warehouse attribute is not provided, the OE searches for a sequence of transformation methods that will generate the desired attribute. If there is no such sequence, and the attribute is not required, its value is set to NULL. If the attribute is required an error is generated, notifying the DBA that another transformation method is required. Because of their complexity, the OE will not attempt to invoke any of the class-methods. Once all the warehouse attributes are defined, the OE uses its SQL interface to generate commands to perform the data transfer.

Incorporating a new data source requires the DBA to describe it, map the source attributes to corresponding abstraction attributes, ensure that all applicable transformation methods are defined, and create the wrapper. The OE then creates the new mediator class, and expands the API as needed. Once a database has been integrated, adapting to minor schema changes often requires only modifying the wrapper to read the new format. Significant changes in the data representation may require the ontology to be modified and a new mediator created.

## 6. Conclusion

DataFoundry is an ongoing research project at LLNL investigating warehousing techniques in dynamic scientific domains. In these domains, the high rate of schemata change makes it difficult, if not impossible, to maintain a warehouse integrating several autonomous data sources using traditional methods. Ensuring the consistency and availability of a data warehouse requires the ability to quickly modify mediators to reflect these schema modifications. This paper presents

DataFoundry's metadata based approach to mediator generation, which is designed to significantly reduce the time and effort required to manage these changes. We expect to have a functional prototype of the OE in place by July 1998, at which time we will begin exploring other uses, such as automatic schema evolution and relational wrapper generation, for the ontology information.

## References

- [1] L. Baekgaard, and N. Roussopoulos. Efficient Refreshment of Data Warehouse Views. UMIACS-TR-96-33, University of Maryland. May, 1996.
- [2] R.J. Bayardo Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyp, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InforSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. May 1997.
- [3] S. Chawather, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proceedings of the ISPJ Conference*. 1994
- [4] T. Critchlow. Schema Coercion: Using Database Meta-Information to Facilitate Data Transfer. Ph.D. Dissertation. University of Utah Technical Report. June. 1997.
- [5] N. Fridman and C. D. Hafner. Ontological Foundations for Biology Knowledge Models. In *4<sup>th</sup> Int'l Conference On Intelligent Systems for Molecular Biology*, pp 78-87, 1996.
- [6] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and Jennifer Widom. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. In *Proceedings of the AAAI Symposium on Information Gathering*, pp. 61-64, Stanford, California, March 1995.
- [7] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. In *Journal of Intelligent Information Systems*, 1997.

- [8] T. Gruber. Ontolingua: A Mechanism to Support Portable Ontologies. Stanford. Knowledge Systems Laboratory. Tech Report KSL-91-66. November 1992.
- [9] T. Gruber. Towards Principles for Design of Ontologies Used for Knowledge Sharing. Stanford Knowledge Systems Laboratory. Tech Report KSL-93-04. 1993.
- [10] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems Techniques, and Applications. In *Data Engineering Bulletin*, June, 1995.
- [11] W. H. Inmon. Building the Data Warehouse. Wiley-QED, 1992.
- [12] D. B. Lenat and R. V. Guha. Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project. Addison-Wesley, Reading MA, 1990.
- [13] L. Liu, C. Pu Y. Lee. An Adaptive Approach to Query Mediation across Heterogeneous Information Sources. In *Proceedings of 1<sup>st</sup> Int'l Conference on Cooperative Information Systems (CoopIS '96)*, Brussels, Belgium, June 1996.
- [14] E. Mena, V. Kashyap, A. Sheth, and A. Illarramendi. OBSERVER: An Approach for Query Processing in Global Information Systems based on Interoperation across Pre-existing Ontologies. In *Proceedings of 1<sup>st</sup> Int'l Conference on Cooperative Information Systems (CoopIS '96)*, Brussels, Belgium, June 1996.
- [15] E. Mena, V. Kashyap, A. Sheth, and A. Illarramendi. Domain Specific Ontologies for Semantic Information Brokering on the Global Information Infrastructure. to appear In *Proceedings of the First International Conference on Formal Ontologies in Information Systems*. Trento, Italy. June 1998
- [16] N. F. Noy and C. D. Hafner. The State of Art in Ontology Design: A Survey and Comparative Review. *AI Magazine*, Fall 1997, pp 53-74.
- [17] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed heterogeneous and autonomous databases. *ACM Computing Surveys*, 22(3):183-236, September 1990.
- [18] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, Procedures, Caching and Views in Data Base Systems. In *Proceedings of ACM-SIGMOD*, Atlantic City, NJ, May 1990.
- [19] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of Disco. In *Proceedings of the International Conference on Distributed Computer Systems*. 1996.
- [20] G. Weiderhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38-49, 1992