2006

A messaging system to handle semantic dissonance

Ashish Rathod

ROCHESTER INSTITUTE OF TECHNOLOGY
Department of Computer Science

A Messaging System to Handle Semantic Dissonance

by

Ashish Rathod

August 07, 2006

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Rajendra K. Raj, Advisor

Zack J. Butler, Reader

Paul T. Tymann, Observer

Signature Block

1) _____ Prof. Rajendra K. Raj

2) _____ Prof. Zack J. Butler

3) _____ Prof. Paul T. Tymann

Acknowledgements

My sincere thanks are extended to:

- Prof. Rajendra K. Raj for his guidance, support and patience throughout the development of this thesis.
- Prof. Zack J. Butler who agreed to be the reader for this thesis.
- Prof. Paul T. Tymann who agreed to be the observer for this thesis.
- Manisha, my wife for motivating me and proof reading this thesis.
- My mother for believing in my ability and providing a constant source of encouragement. My family members who supported and motivated me to complete this thesis.
- Paul Rathbun, my manager for allowing me to take time off work to complete this thesis.

TABLE OF CONTENTS

A Messaging System to Handle Semantic Dissonance

Abstract

Enterprises have been compelled to share their data internally and externally, but creating a consistent view of enterprise data has been challenging. Within a typical enterprise, each division uses its own domain specific data model and schema, and different enterprises obviously use their own data models and schema. Integrating these diverse data models and schemas, which have both syntactic and semantic differences, tends to be complex, slow, and inaccurate. Syntactic differences, i.e., differences in names or layout, have received substantial attention in research. *Semantic dissonance* simply means that the structure may be similar (or even the same) but the ***meaning*** associated with the attributes that define each structure are different, has received less attention in the world of practical software development.

A practical messaging system for handling semantic dissonance has been developed. The system utilizes the Resource Description Framework (RDF) and SOAP XML Messaging Specification. It is implemented using Jena, a Java API for RDF, and the Apache SOAP, an Open Source SOAP server and client. This report describes the messaging system, its implementation, its strengths and limitations in handling semantic dissonance.

1. Introduction

With increased internal and external cooperation, enterprises have been compelled to create a consistent view of their proprietary data, and need to redesign data in order to make it uniform and shareable. Creating a consistent view of enterprise data has been challenging. Within a typical enterprise, each division uses its own domain specific data model and schema, and different enterprises obviously use their own data models and schema. Integrating these diverse data models and schemas, which have both syntactic and semantic differences, tends to be complex, slow, and inaccurate.

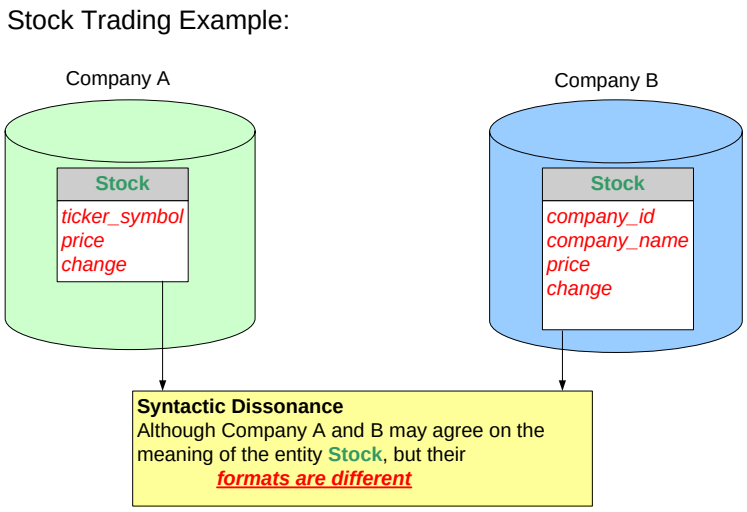

Figure 1 Example of Syntactic heterogeneity

Syntactic heterogeneity means that the structures are different in format but have similar/same in meaning. This form of dissonance can be resolved by adopting a canonical schema/model. An example (shown Figure 1) would be of two stock trading companies (Company A and B) having different formats to represent company/stock information. Company A defines the *attributes* for the entity stock as (ticker symbol, price, change) whereas Company B defines the entity stock as (company id, company name, price, change). Although Company A and B may agree on the meaning of the entity stock their formats are different.

Semantic dissonance simply means that the structure may be similar (or even the same) but the ***meaning*** associated with the attributes that define each structure are different. A major factor that contributes to semantic dissonance is *design autonomy*, which means that different enterprises build their own design or models to represent the information managed by their systems. This modeling problem finds its roots mostly during the design phase of a database schema. Different user groups and designers adopt their own viewpoints when modeling the same information. An example to illustrate the semantic dissonance problem is as follows:

Consider an attribute price of *relation* stock_price in database db1 that describes the buying price of stock without trading charges. Consider an attribute by the same name price of relation trade_price in database db2 that describes the buying price of stock with trading charges. The assumption here is that the attribute price has the same syntax in both databases. An attempt to compare relations db1.stock_price.price and db2.trade_price.price are misleading since these relations are semantically heterogeneous.

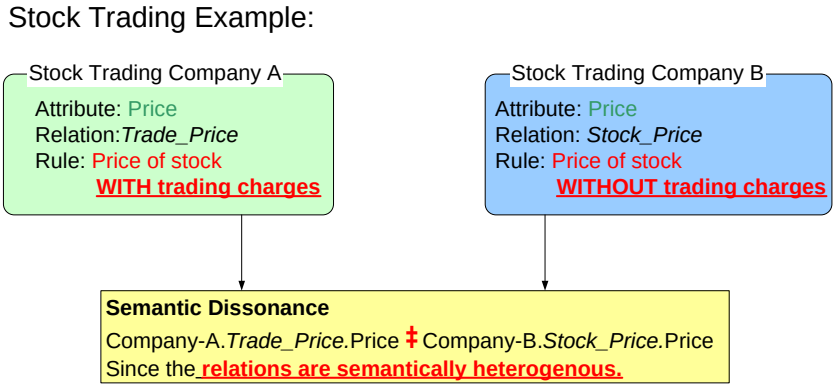

Figure 2 Example of semantic dissonance

Due to semantic dissonance, many enterprise applications reference the same data but associate different meaning or semantics with it. The problem is further exacerbated by the fact that application databases are not synchronized. Integrating such applications, involves extracting semantics and manually resolving inconsistencies. Although this

problem is well known in large number of enterprises, tools and techniques to handle this problem are only in research or early prototype stages.

The two major industry initiatives to handle semantic dissonance are: Enterprise Application Integration (EAI) and Enterprise Data Integration (EDI). The EAI approach handles semantic dissonance by using the concept of middleware, whereas EDI approaches the problem by trying to derive a canonical schema by using the concepts in data warehousing. A new industry called Enterprise Information Integration (EII) has developed in the late 1990's. The purpose of this effort is to provide tools for integrating data from multiple sources without having to convert to a standard format [12]. However EII has several challenges, the toughest problem is still related to metadata management and handling semantic dissonance.

This thesis represents an effort to investigate the techniques to handle semantic dissonance using an object oriented framework called the Active Collections Framework (ACF). ACF enables enterprises to share data in near real time. ACF in its initial form did not deal with semantic dissonance issues. The use of this messaging system extends ACF to handle semantic dissonance. The system design uses Message Transformation Patterns in [13], Resource Description Framework (RDF) [24] and SOAP XML [25] Messaging Specification. It is implemented using Jena, a Java API for RDF, and Apache SOAP, an Open Source SOAP server and client. This research is aimed at investigating the use of RDF as a data integration technology, previous RDF applications typically have focused in areas of search technology or semantic web but not widely used in handling semantic data integration.

This report describes the messaging system, its implementation, and its strengths and weaknesses in handling semantic dissonance. The rest of this section, briefly explains Active Collections Framework (ACF). Chapter 2 explains the hypothesis, Chapter 3 explains the design and Chapter 4 explains the implementation of the messaging system

to address semantic dissonance. Chapter 5 presents the analysis of this implementation and related work in this area. Chapter 6 describes the current state of this work and concludes with a summary of the contributions of this report.

ACF [20] provides a framework for building distributed applications that share data in near real time. A key concept proposed by this framework is that of an Active Collection, which reflects the collection of ACF objects based on a business rule or predicate. The ACF Object is a representation of data stored in underlying data store. Each ACF object in the collection is checked periodically to ensure that it satisfies the rule/predicate specified. The ACF objects are dynamically added or removed from the Active Collections based on the business rule/predicate.

2. Problem Definition and Hypothesis

Semantic dissonance (sometimes referred to as semantic heterogeneity) is a term used to describe "*differences* in meaning, interpretation, and intended use of same or related data" [8]. Semantic dissonance results from several reasons. One major reason is *design autonomy* [27] in which each architect or developer may model the application or database based on their background knowledge and viewpoint of the business requirements. Another reason is *association autonomy* [27], in which applications determine the information they need to share with other systems. Other reasons include *communication autonomy* [27], i.e., a system decides if it is necessary to communicate with other systems, and *execution autonomy* [27], i.e., the decision to execute local operations without interacting with external systems and the order in which to execute the operations.

ACF, the object-oriented framework for near real time distributed data sharing mentioned in Chapter 1, does not deal with data dissonance. It simply assumes that the data is semantically standardized [21] because data is shared between closely designed applications. This thesis represents an effort to investigate techniques to handle semantic dissonance in the context of ACF.

Based on several years of research, the following techniques have been used to handle semantic dissonance [16]:

- *Schema Mapping/Matching approach:* constructs mappings between semantically related sources. The drawback with this technique is the inherent complexity of mapping related sources that requires prior resolution of conflict. In addition, it is difficult to maintain these mappings in the event of changes.

- *Intermediary approach* uses an ontology which is a meta-model that contains knowledge for a specific domain. The drawback with this approach is that developing and maintaining such a knowledge store is practically not possible.

- *Query-Oriented approach* uses query languages that are declarative logic-based. The idea in this approach is to develop queries capable of spanning multiple systems. The drawback with this approach is that it requires the developer to have in depth understanding of the systems against which the queries are executed.

This hypothesis states that:
The Resource Description Framework together with a Messaging System can be used to partially automate integration of semantically heterogeneous application systems.

A fundamental assumption with this approach is that an application's data and/or object model can be expressed in RDF Schema format. Typically an application represents its data model using E-R diagram and object models using UML diagrams. Tools for converting application models (both E-R and UML) to RDF Schema are available but I did not perform an in-depth analysis of these tools, since it is beyond the scope of this report. Another assumption is that the service provider publishes their canonical model and associated rules using a well known registry service.

The use case explains the steps involved in this approach:

1. The client converts their model to RDF Schema format.
2. The client makes a request to the messaging service to share its object by providing its model and object.
3. The messaging service ***matches*** the client's model to the service's model.
    a. The match is performed between RDF Schema of the service and client on the basis of subject (represents entity/class), predicate (represents

relationships/attributes) and object (represents value of the relationship/attribute) defined in each schema.

4. If the match between the client and service model is successful, then request to share the object is processed, otherwise the request is declined by the service by specifying the elements that did not match.
5. The request log for these rejected requests can then be reviewed by the integrator to manually handle the elements causing semantic dissonance.

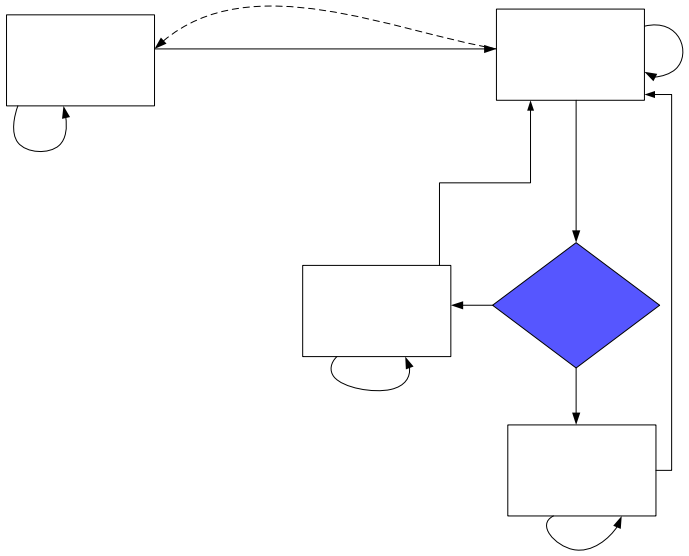

Figure 3 Interaction diagram for the Hypothesis

This approach can be further explained with a help of an example. Consider a stock trader with an application which provides a price of stock *with trading charges*. A centralized stock trading service accepts price quotes from different stock traders and provides these prices to users that are interested in these prices. The centralized stock trading service requires stock traders to send the price quotes *without trading charges.*

The attributes defined on the company object in a stock trader's database are as follows:

Attribute	Description
Change	The changes in price for stock
Price	Total price of the stock
Name	Company name
Ticker symbol	The symbol assigned to the company by the stock exchange authority

Table 1 Data Definition for Company object in a stock trader's database

The total price is assumed to include all the charges that include trading charges in order for the subscriber to purchase the stock. However this assumption may not always be true and may differ from one stock trader to another. If the stock trader's application, sends the data from the Company table as shown in Table 1 then the centralized service cannot determine if the stock trader price is with or without trading changes. In such a scenario, the service cannot *assume* that the stock trader's price does not contain the trading charges. To address this problem, the stock trader's application sends their object model to the service as shown in Figure 4. The service can then *match* the object model to the centralized service model to determine if the price of the stock is *with* or *without* trading charges. The *match* depends on the fact that the stock trader's model contains an entity/class called TradingCharges.

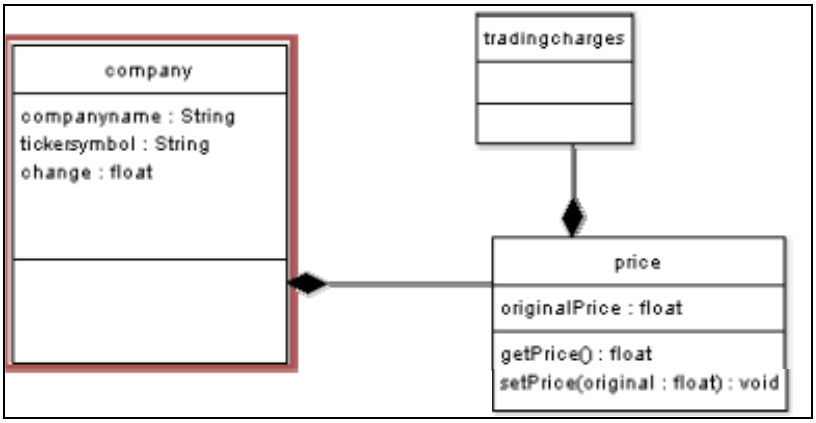

Figure 4 UML Class diagram for Stock Trader Application

The UML class diagram in Figure 4 shows that the *Class* Company has three *attributes* and one composition *relationship* to the *Class* Price. It also shows that *Class* Price has a composition *relationship* to *Class* TradingCharges. Typically, modeling languages like UML allow developers and architects to represent classes, attributes and relationships. However it does not provide the logic which may be embedded in the source code that is used to associate additional semantics to a specific attribute. On the other hand, a programming language like Java provides the source code (syntax) which can be reverse engineered to understand the semantics associated with a specific attribute.

```
1   Class Price{
2           Float originalPrice;
3           TradingCharges tc = new TradingCharges();
4           Float getPrice(){
5                   float price = originalPrice + tc.getCharges();
6                   return price;
7           }
8           void setPrice(float original){
9                   originalPrice = original;
10          }
11  }
```

Figure 5 Source Code of Class Price

In Figure 5, the statement on line 5 shows that the price of the stock contains trading charges.

Due to above mentioned problem, it is possible to only partially automate the integration of different applications to a service. In order to fully integrate applications to the service, developers and architects need to review the requests rejected by the service to determine which attributes or relationships caused the semantic dissonance and then manually resolve the mismatch, since they can read the source code to determine the exact rule/logic that provides the complete context for the data being integrated.

3. Design

A common problem faced while integrating enterprise applications is the use of different programming languages for specifying business logic [13]. This makes extraction of application logic especially hard since the integrator needs to know several programming languages in order to decipher the logic encoded within the application. This influences the amount of time and money required to integrate the applications to a great extent.

To integrate application systems, an integrator can choose from the following integration patterns [13]:

- In *File Transfer* [13], applications that need to share data, export their data to a file in a well known format; applications that need to use this data consume it using the well known format.
- The *Shared database* [13] approach uses a common database to specify shared data, similar to a data warehousing format.
- The *Remote Procedure Call* [13] uses a concept in which applications that need to share data expose their methods to consumer applications. The consumer application can access the shared data by invoking the application's exposed method and passing in the correct parameters.
- The *Messaging* [13] provides a flexible framework to share data between applications by using a common messaging system. A messaging system consists of five components: channel, router, message, translator and endpoint. An explanation of these components is provided in the following paragraphs. Messaging was selected because it provides a method to integrate applications independent of programming language and platform. Messaging also provides loose coupling between applications and service which simplifies integration of large enterprise applications.

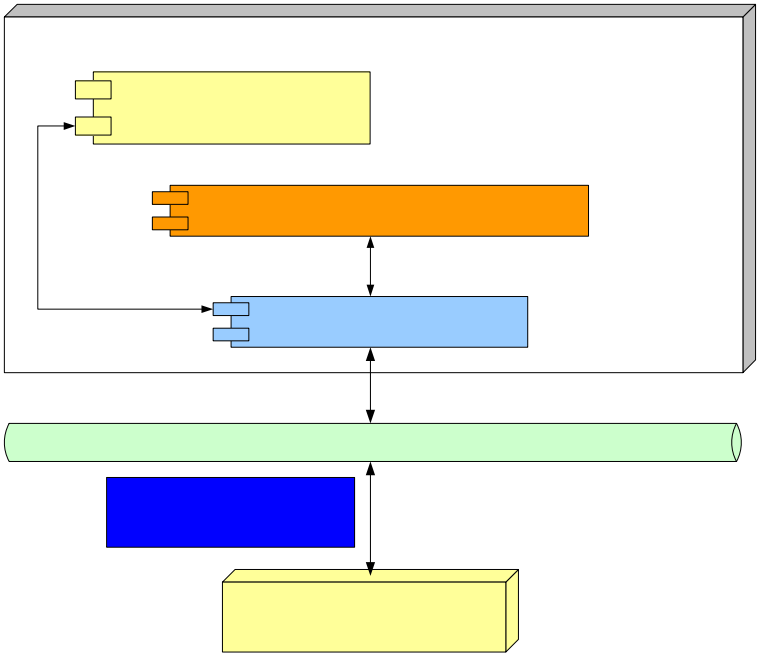

Figure 6 ACF Messaging subsystem components (Adapted from [13])

The design of the ACF Messaging subsystem as shown in Figure 6 contains the following five components:

- ***Channel*** [13]: Applications use a transport or channel to send and receive data. Client server applications typically use the TCP/IP transport to transmit and receive data. Web based applications use HTTP which is a protocol layered on top of TCP/IP. This design uses the Publish-Subscribe channel [13] as it allows clients to publish and subscribe messages to the messaging service in an asynchronous manner.

- ***Message*** [13]: It is the smallest indivisible form of message request that an application can transmit to a channel. Proprietary message formats are not sufficiently flexible to enable semantic integration [21]. This design uses SOAP, which is an industry wide standard for XML Messaging over HTTP. The advantage of using SOAP over other frameworks (like CORBA, Java RMI) is

that it uses XML to specify request-response messages. In simple terms SOAP which is an acronym for Simple Object Access Protocol, is an object protocol that allows applications to send and receive messages in a platform neutral manner. It specifies the format of messages along with a set of encoding rules. It consists of three parts: the envelope, header and body.

The SOAP specification as shown in Figure 7 defines three major parts [4]:

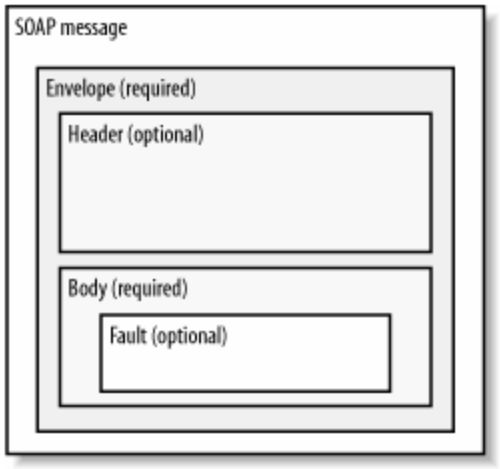

Figure 7 Main elements of a SOAP Message (Adapted from [28])

The Envelope section represents the root element for a SOAP message; it is used to specify the version of SOAP being used. The Header section is optional and contains two attributes for applications to specify different types of application specific data. The first attribute is called *Actor*, where an application can specify the recipients for the SOAP message. The second attribute is called *MustUnderstand*, indicates if the SOAP header is optional or mandatory. The Body section is used to specify the class, method, parameters and return values to service provider's class.

For example, a stock trader's application must follow the rules agreed upon by the stock trading service that it uses to send stock quotes. In case of an error the body section will contain a fault element indicating the reason for the failure.

- ***Router*** [13]: provides message encoding mechanisms so that a message is delivered to the receiver. Message filter pattern [13] provides a mechanism to route message requests based on the content of the message. A filter ensures that only messages that satisfy that criteria get delivered to the recipients.

- ***Translator*** [13]: Message transformation is required because applications may not agree on the meaning or format of the shared data. Message transformation can be divided into two types: schema transformation based on differences in format and behavioral/semantic transformation based on differences in meaning or logic.

  Canonical Data Model (also referred to as Ontology) [13] helps identify the differences in both data types and meaning. However defining a Canonical data model requires both knowledge and experience about the problem domain. It is usually developed by a combination of visual meta-modeling editors and human expert architect/modeler. There is a considerable amount of research being performed to develop a fully automated mechanism to develop canonical models, but most researchers have led to the conclusion that the generation of a canonical data model can only be partially automated.

  Resource Description Framework (RDF) [24] was selected to handle semantic dissonance because it provides the framework to specify metadata about objects. It also provides inference mechanisms to interpret the rules associated with them. RDF has a large and active audience devoted to developing the specification as well as tools that help test the validity of the specifications. RDF

uses a triple format to specify an assertion about a subject. The RDF triple consists of {subject, predicate, object}. The subject is a resource, predicate refers to a particular relationship to the subject and the object is the actual value associated to the predicate.

RDF provides a model for representing metadata similar to what Relational data model does for relational database. It allows programmers to develop queries against the defined RDF Model in a platform independent manner. A collection of RDF statements (also called triples) is called a RDF graph. A RDF Graph is a directed graph as shown in Figure 8.

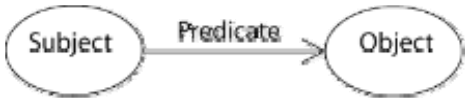

Figure 8 RDF Triple as a graph (Adapted from [23])

RDF Concepts and Abstract Syntax [24] states that:
"Two RDF graphs *G* and *G'* are equivalent if there is a bijection *M* between the sets of nodes of the two graphs, such that:

1. *M* maps blank nodes to blank nodes.
2. *M(lit)=lit* for all RDF literals *lit* which are nodes of *G*.
3. *M(uri)=uri* for all RDF URI references *uri* which are nodes of *G*.
4. The triple *(s, p, o)* is in *G* if and only if the triple *(M(s), p, M(o))* is in *G'*

With this definition, *M* shows how each blank node in *G* can be replaced with a new blank node to give *G'*."[24]

This design uses Jena [3] an open source Java implementation of the RDF specification. It provides a programmatic environment for RDF, RDFS, OWL,

and a rule-based inference engine. Since the client and service schemas are expressed as RDF Graphs, operations such as comparison for equality, difference, union and intersection between the two models can be performed using the Jena API.

- ***Endpoint*** [13]: It is either a producer or consumer of messages within a messaging system. The messaging service provides a messaging client (API or agent) in form of an agent or API since applications typically do not have native support in order to connect to the messaging service. The messaging client allows applications to integrate with the messaging service in a seamless manner since the messaging client is provided in the client's programming language and platform.

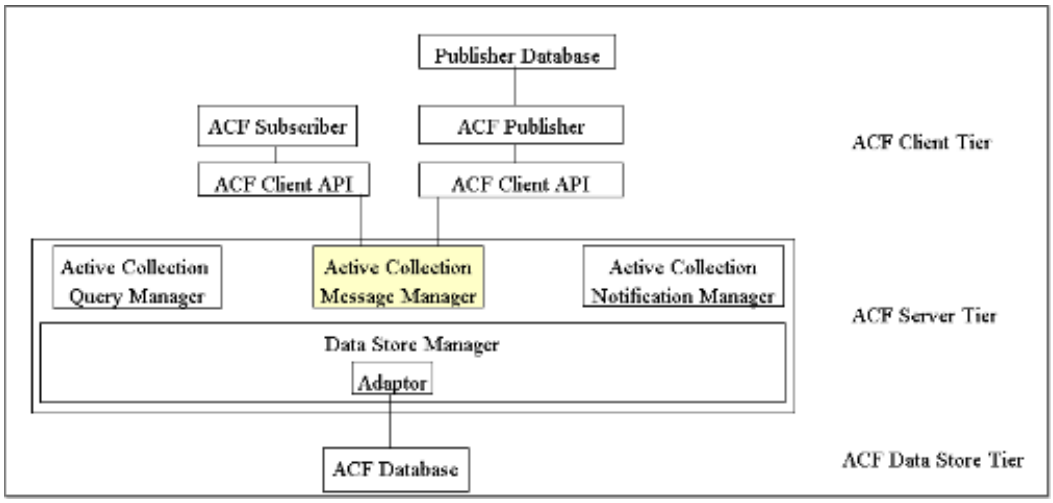

Figure 9 Logical Architecture of ACF (Adapted from [20])

Figure 9 depicts an extension to ACF's logical architecture. This research is aimed at extending ACF to include an additional component called Active Collection Message Manager. The purpose of this component is to provide an open messaging transport between ACF Clients and Service as well as to handle semantic dissonance within the ACF framework. ACF Clients i.e. Publishers and Subscribers invoke ACF Services using the ACF Client API.

4. Implementation

The focus of this implementation is to develop a messaging system, which uses Apache SOAP API [31] (an implementation of the SOAP specification) and Jena API [3] (an implementation of the RDF specification) to explore techniques to identify and reduce semantic dissonance.

This implementation uses the core ACF implementation developed at Rochester Institute of Technology by Bhaskar Gopalan [9]. The implementation in [9] uses a proprietary messaging format for communication between the ACF Client and Service. Another major assumption of the implementation in [9], is that all publishers to the ACF Service agree on the meaning of data, however this is not true in most cases.

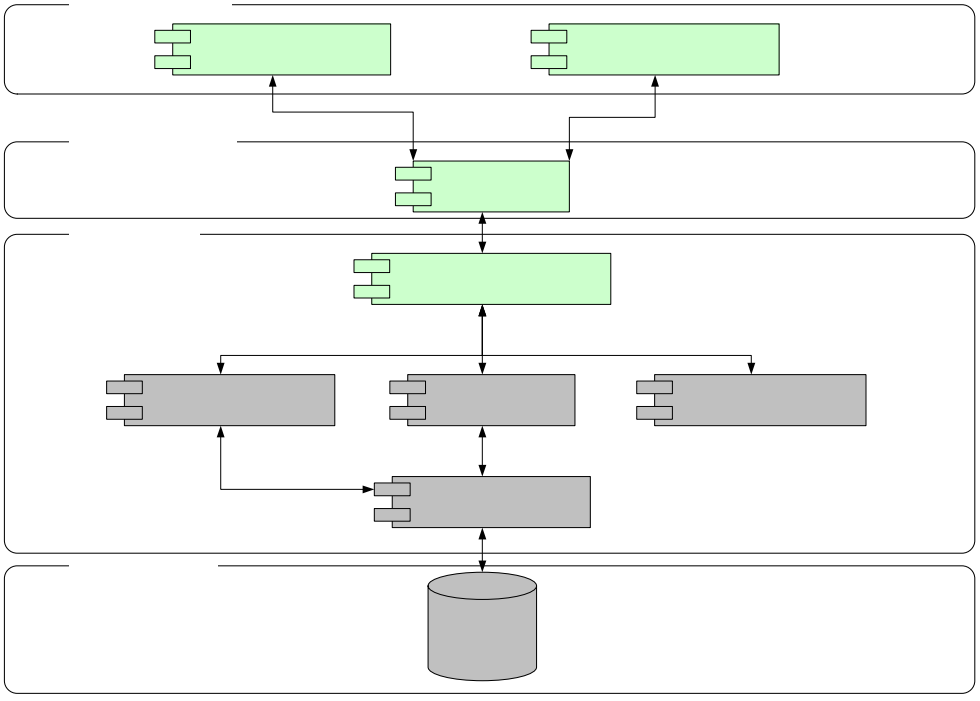

Figure 10 System Architecture of ACF Messaging Subsystem components (Adapted from [9])

Figure 10 depicts the components within the Active Collections Framework. It consists of a set of managers which use the core business objects. A brief explanation of these components is provided below.

- *ACFMessageManager*

  This component is responsible for accepting requests from ACF Clients i.e. Publishers and Subscribers. It delegates the processing of these requests to the corresponding processing managers. To handle semantic dissonance this implementation uses the Jena API which is an open source RDF implementation. It provides extensive features to perform match operations on RDF Graphs. Figure 8 shows an example of RDF Model of the Company Schema for the ACF Service. The schema makes statements/assertions about the company object. For example, *company* (subject) *HAS* (predicate) *price* (object). The object price also has statements associated with it, which states that *price* (subject) *IS* (predicate) *CLASS* (object).

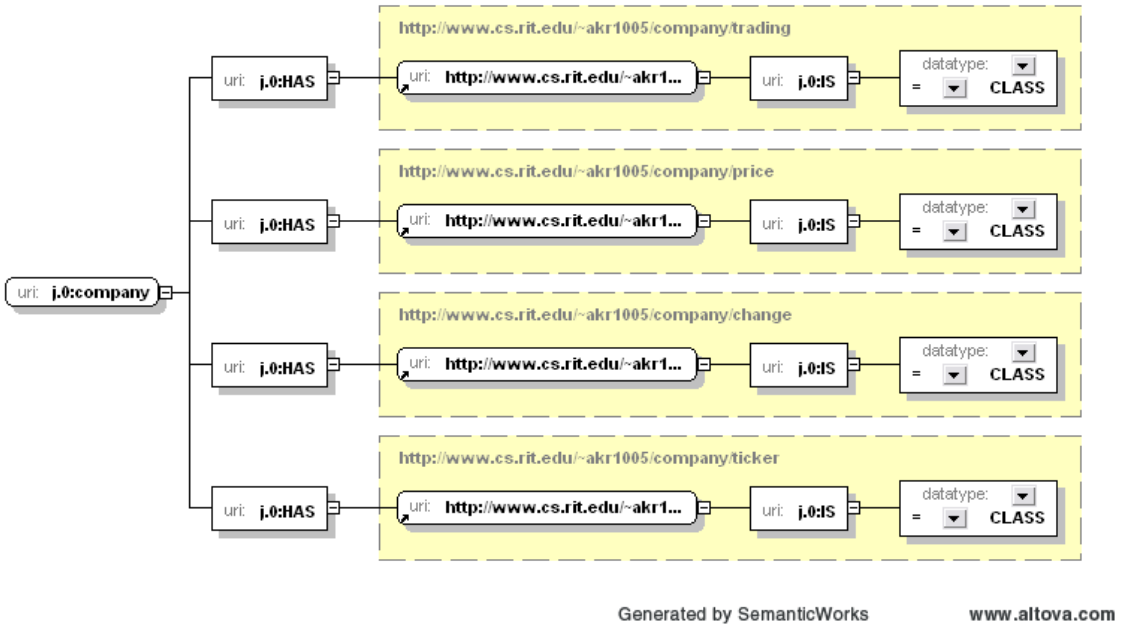

Figure 11 RDF Model of the Company Schema

- *SOAP Server*

  In order to overcome the constraint of using proprietary message format, this approach re-implemented the transport between the ACF Clients and Service using the Apache SOAP API. The ACF service is uniquely identified using the urn *urn:acf:acfservice*. The SOAP server routes requests from the ACF Clients to the provider class called *ACFMessageManager*. The services provided by the ACF are exposed using Apache SOAP server. Figure 12 shows a snapshot of the Apache SOAP Admin Console that displays the deployment descriptor for ACF Web Service.

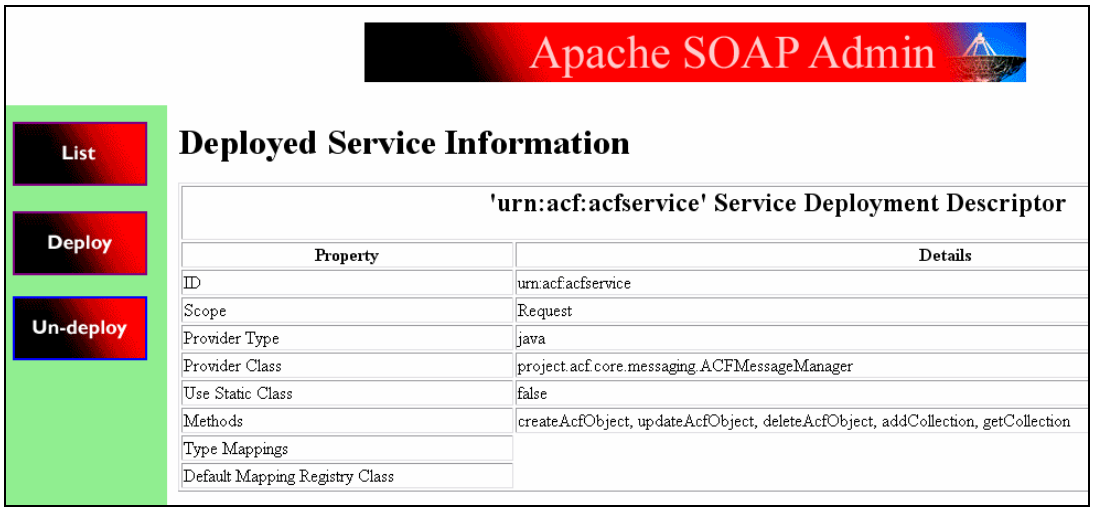

Figure 12 ACF Web Service Deployment Descriptor

- *Publisher and Subscriber SOAP Client*

  ACF Clients use standard SOAP libraries to send and receive requests to and from the ACF Messaging Service. They invoke the service using the urn, Provider class and specific method depending on the service required.

A typical usage scenario of the ACF messaging service would be as follows:

1. The ACF client invokes the ACFMessageManager using a Web service call which includes the type of service desired.

2. The ACFMessageManager deciphers the type of request and delegates the requests to the appropriate processing manager. If the request is a create *Active Collection* object from a Publisher client, the ACFMessageManager forwards the request to the RDFMessageReasoner to determine if the publisher's schema matches the canonical model/schema. At this point, service can determine if there is semantic dissonance between the publisher and the ACF Service models.

3. The *QueryManager* is invoked to handle requests from publisher clients for creating, updating or modifying *BaseACF* objects. Requests from subscriber clients are delegated to the Collections Manager.

There are two major components to the messaging system implemented in context of the ACF:

***ACF Messaging subsystem*:** This subsystem is responsible for comparing the relationships in the publisher's model with the enterprise/domain model. The classes of this subsystem use the Inference engine provided as a part of the Jena along with the Apache SOAP API to perform SOAP related functions. The assumption here is that the publisher's model is expressed in RDF Schema format. Figure 13 shows the classes associated with the ACF Messaging Subsystem.

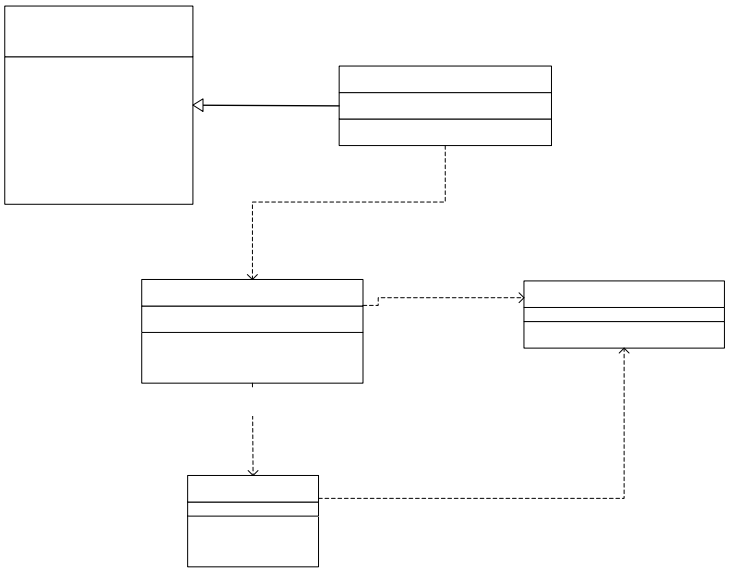

Figure 13 ACF Messaging Subsystem Class Diagram

I implemented the ACF Messaging Subsystem by developing an interface that defines all the services that ACF offers. These include methods to parse XML messages and normalize them based on a standard XML or RDF Models. It includes the following methods:

- *createACFObject()* matches the publisher's schema with the enterprise/domain schema. If the schemas match then it inserts the object into ACF Database.
- *deleteACFObject()* deletes the publisher's object from the ACF database.
- *updateACFObject()* updates the publisher's object in the ACF Database.
- *addCollection()* creates a new Active Collection based on predicate provided by the Subscriber.
- *getCollection()* retrieves the latest Active Collection based on the predicate provided by the Subscriber.

- *removeCollection()* removes the Active Collection from the Collection repository.

The ACFMessageManager class acts as a "controller" for the ACF Messaging Subsystem by validating the requests made to this subsystem. In most cases it will delegate the processing of the request to a concrete implementation within the model for the ACF messaging subsystem.

This RDFMessageReasoner class is responsible for *matching the publisher's model with the service's canonical model.* If it finds that there is an inequality, it sends the request to the Normalizer class which tries to handle the semantic dissonance by identifying the elements where the inequality occurs. If it is unable to detect the elements causing the inequality, it sends a message to the Publisher indicating the attributes that are not equivalent to those stored in the canonical model.

- *modelEqual()* – This methods takes the publisher's model as input and determines if the models are equal.
- *modelDifference()* – This method takes the publisher's model as input and provides the attributes that are different from the enterprise model.

The Normalizer class is responsible for trying to handle the semantic dissonance by detecting the elements where the inequality occurs. It provides the following methods:

- *union()* – This method takes the publisher's model as input and creates an union of the publisher's model with the enterprise model and returns a RDF Model as a result.
- *intersection()* – This method takes the publisher's model as input and creates an intersection of the publisher's model with the enterprise model and returns a RDF Model as a result.

The RDFOntologyModel class stores the RDF model which represents the Canonical data model for the ACF messaging Service.

- *standardCompanyModel()*: The RDFReasoner and Normalizer classes use this method to retrieve the canonical RDF Model for ACF messaging service.

***ACF Client API*:** This API is responsible for converting the publisher's model and data into the RDF format and providing the subscriber clients, data in XML format based on the query setup during subscription.

The subscriber client interface consists of methods to retrieve the Active Collection and define rules/predicates for create ACF Collection. This includes methods to

- *addCollection()* – creates new Active Collection
- *getCollection()*- retrieves the latest Active Collection
- *removeCollection()* - deletes an Active Collection

The publisher client interface consists of methods to create, update and delete the ACF Object.

- *createACFObject()* –creates a new ACF Object.
- *deleteACFObject()* – deletes an ACF Object.
- *updateACFObject()* – updates an ACF object.

Figure 14 illustrates how semantic dissonance is handled using the ACF Message subsystem.

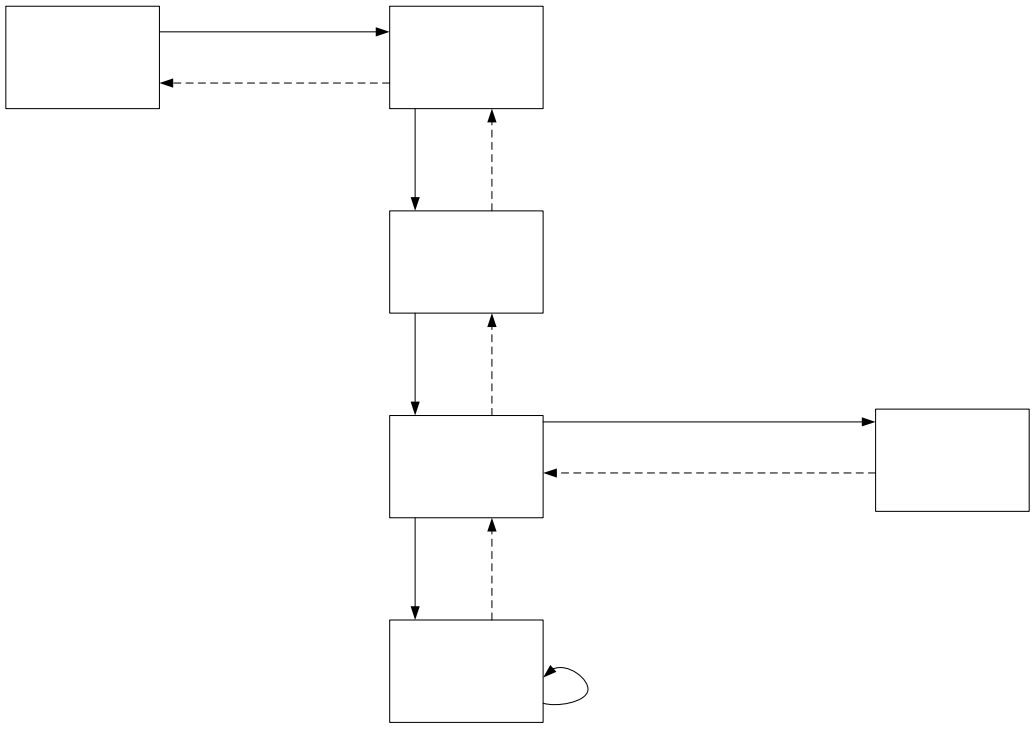

Figure 14 Interaction Diagram for ACF Messaging Subsystem

5. Analysis

A major lesson learnt was that you cannot fully automate the resolution of the semantic dissonance problem. The schema mapping approach discussed earlier is insufficient to address the semantic dissonance problem. After reviewing several papers, I realized that while trying to integrate enterprise applications, the application code contains ***logic*** that is not accounted for during the integration exercise, which developers and architects need to handle manually. Meta-data plays a major role in handling semantic dissonance, however industry accepted frameworks and tools required to manage meta-data across an enterprise are not mature enough [12].

Based on this study, I found that there is no single approach sufficient to solve this problem. Although research in this field has been carried out for over twenty years, the formal complexity of different types of semantic heterogeneity algorithms has not yet been determined. During the study, I found that there are a number of different approaches taken by different research groups, no single industry-wide effort exists to solve this problem.

I found that E-R diagrams, UML diagrams and other meta-models do not capture all the semantics associated with the data. Rule languages similar to declarative languages like LISP, PROLOG which are based on the first order of logic are required to specify the semantics associated with data in more detail. I also found that there is a need to study security models that establish *trust* between services and applications, since applications need to share their model/schema during integration.

Test Results

In order to test the implementation I developed three schemas. The schema S represents the stock trading service's schema which defines the set of standard set of elements associated with a company object. Schemas S1 and S2 represent the models for the stock trader's schemas that need to use the stock trading service. In the test I used the messaging system to perform compare, difference and intersection operations between the service Schema S and stock trader schemas S1 and S2 (Appendix D Test Data contains the RDF representation of these schemas). The results of the tests are shown in Table 2.

Operation with Service Schema S	***compare***	***difference***	***intersection***
Stock Trader Schema S1	Not Isomorphic, structures are not equivalent	Schema S1 contains additional class *trading*	Result of intersection between Schema S and S1 yields the following set of common elements: {ticker, price, change}
Stock Trader Schema S2	Isomorphic, structures are equivalent	No differences found since the schemas are isomorphic.	ALL elements in schema S are present in schema S2 so the result of intersection is: {ticker, price, change}

Table 2 Test results

Strengths

This approach minimizes the manual integration of application model/schema with the enterprise model/schema. A standards based approach to handle semantic dissonance was taken by using SOAP and RDF specifications. I believe this will enable this design to be interoperable and extensible in the future.

Limitations

Potential performance problems that could occur as there are several steps (parsing RDF/XML) involved in adding a new object using the ACF Messaging Service. The ACF Messaging subsystem uses the HTTP as its underlying protocol. Since the HTTP protocol is stateless the ACF Messaging System is unable to notify clients by performing a callback.

Related Work

ebXML [7] has been proposed as an electronic business standard for a business of any size to do business using the internet. It provides a syntactic framework, however the generation of contracts (semantics) i.e. Collaboration Protocol Agreement (CPA) and Collaboration Protocol Profile (CPP) needs to be provided by the enterprises, which is similar to what would be required in case of integrating with other enterprises in any case. The visible advantage of using ebXML would be to use a standard schema. The problems associated with CPA formation process are explained in [26].

OpenCyc [15] is an open source version of Cyc technology, the world's largest general knowledge base and common-sense reasoning engine. This initiative provides the tools which enable designers to develop generic models also known as ontologies for a given problem domain. The drawback for this approach is the difficultly in integrating these tools with XML like messaging infrastructure.

“A deductive database system includes capabilities to define rules, which can deduce or infer additional information from facts stored in the database.”[8] It uses two main types of specifications: facts and rules. Facts are statements that can be asserted to be true. Rules are conditional statements which can be evaluated to provide a boolean result i.e. true or false. The drawback of these systems is that they have complex language rules

and based on the first order of logic languages like LISP, PROLOG which are difficult to learn and apply in mainstream programming.

Aslan & McLeod [1] present a technique to add metadata to local schema in order to eventually evolve a global schema. Hakimpour & Geppert [10] provide a technique to merge ontologies based on similarity relations among concepts in different ontologies. Rahm & Berstein [19] conducted a survey of automatic schema matching approaches developed so far and proposed taxonomy to simplify the type of matching schemes for future use by programmers and researchers.

Ram & Park [16] propose CREAM, a semantic interoperable framework using an 8-tuple framework, but cautioned that their results are subject to interpretation. Stumptner, Schrefl & Grossman [29] proposed a meta-class based approach to behavioral integration however they are still in the process of validating this approach.

Most of the approaches described above provided partial resolution to the semantic dissonance problem. Each of these efforts approached the problem from either a database perspective that focused on schema mapping/matching or development of agents and ontologies.

6. Conclusion and Future Work

Semantic dissonance is one of the toughest problems in Data Integration. The current state of the problem is that large enterprises need to apply more than one approach to integrate their systems which is time consuming and costly. Several approaches have been proposed in the past two decades however no solution addresses all aspects of the problem. Research in academia and industry has resulted in identifying two major approaches to addressing the problem: EAI and EDI.

This research explains an approach to handle semantic dissonance within the Active Collections Framework which is based on message based application integration concepts in EAI. I developed a messaging system that uses RDF and SOAP to handle semantic dissonance. The key aspect of this messaging system is that it allows application to validate their application model at runtime with that of a service. The service is capable of performing compare, difference and intersection operations between the application and service models to detect semantic dissonance. A prototype of this messaging system was built using Jena and Apache SOAP API's and successfully tested using an example of a Stock Trading system.

It was observed that semantic dissonance can be detected and resolved only to a limited extent, it cannot be completely automated. In many cases human intervention is still required to resolve semantic dissonance. During this research I studied and evaluated ebXML, Opencyc and deductive databases as possible alternatives, however each approach had some drawbacks. A common thread among the solutions is the use of metadata to handle semantic dissonance. However there are several languages used to express metadata which include E-R, UML, XML Schema etc… which requires mapping between models when different applications need to be integrated. I also found that there is need to use declarative logic based languages like OWL to fully express application logic as metadata. My decision to use RDF was based on the fact it provides

a core features to represent metadata in a platform independent manner. In addition RDF has been declared as one of the cornerstones for the Semantic Web and has a large developer community implementing the specification in different languages.

Based on my work so far, I believe the following activities will help extend the concepts that have been implemented:

- Investigate tools for conversion from ER to UML to XMI to RDF Schema.
- Replace Rule related components within ACF with open source rule engines that are more robust and provide a wider range of rule specification syntax.
- Study to determine asymptotic complexity of semantic heterogeneity algorithms.
- Improve the Client API functionality by developing agents to convert client application logic into Web Ontology Language - OWL [2] to help provide better metadata for inference.
- Evaluate security models for establishing trust between ACF Clients and Service.

Based on the lessons learnt from trying to handle the semantic dissonance problem, further research must continue to address subsets of this problem domain in order to build an intelligent infrastructure. This will enable the next generation of applications and services that can adapt to changes based on their ability to infer meaning from metadata.

Appendix A - *Glossary*

Query Manager – This component is responsible for creating, updating and deleting ACF objects based on predicates specified by the ACF subscriber clients.

Collections Manager – This component manages the Collections Repository. All operations on the repository have to be made through this class. This class is notified by DBMonitor on any database changes. All the collections in the repository are updated accordingly.

Notification Manager – This component notifies the client about any changes in the database by writing information about the changed record to a serialized bean.

Database Manager –This component provides an interface between the application and the underlying database by providing methods to create, update, delete, list ACF objects.

BaseACF – This class represents base class of all ACF Objects. It contains methods to retrieve the parameters, columns for database operations on the ACF object. All the ACF Objects are serializable and map to a database package name. Each object has a unique Id and also a owner Id which denotes the owner (creator) of this object.

Active Collection – Active collection of a particular subscriber. It stores members both in the form of ordered List. Every time the collection is modified, the members are sorted.

DBMonitor – Monitors the ACF database periodically for any change in the data. The table to be monitored and the period of monitoring are retrieved from Configuration.

Rules Engine – Used to run rules of an active collection against a given ACF object. The rules are nothing but the query parameters that make up the collection. The given object is checked to be valid against each rule.

SmartBean - A serializable bean that encapsulates information about the member modified in the active collection. This bean encapsulates information changed and also the operation, like insert, remove, update, etc.

Appendix B –List of Tables and Figures

Tables

Figures

Appendix C – User Manual

Project environment

The following development environment was used to build and test ACF message subsystem.

- Languages: Java(jdk1.3), XML
- Tools & API:
  - IBM WebSphere Studio Application Developer
  - TOAD
  - Jena API
  - Apache SOAP API
- Database: Oracle 8i
- Operating System : Windows XP professional
- Application Server: Websphere Test Environment 5.0

Project Setup

It is assumed that the reader has access to the development environment specified in the 'Project Environment' section.

The project CD contains the following files:

- ACF.ear – J2EE enterprise application resource. It contains the java source code and configuration files required to run this implementation.
- tables.sql – Database scripts to create the ACF database schema
- Supporting Libraries
  - Apache SOAP API and dependencies
  - Jena API and dependencies

Steps to setup

1. Create a new database acf, with username and password as “acf”.
2. Connect to acf database using TOAD and run the tables.sql script using TOAD. This will create the database objects required.
3. Start IBM Websphere Studio Application Developer.
4. Import ACF.ear and add required libraries to Java build path
5. Create a new Websphere 5.0 test environment ACF Server.
6. Create an oracle data source for the project using ACF server created.
7. Add the ACF Project to the ACF Server.
8. Start the ACF Server.

Steps to Test

1. ***ACF Server***
    1. Open a web browser and type in the URL: http://localhost:9080/ACFWeb/admin/index.html
    2. The Apache SOAP Admin page is displayed. Click on the List to view the services deployed.
    3. Click urn:acf:service option from the list of deployed services.
    4. The methods column in the list displays the methods that are exposed by the ACF Web Service.

2. ***ACF Publisher***
    1. Open a web browser and type in the URL: http://localhost:9080/ACFWeb/PublisherServlet?request=<request_type> <request type> can take one of the following values:
        1. create
        2. update
        3. delete
    2. The status of the web service request is displayed.

3. ACF Subscriber

1. Open a web browser and type in the URL:
   http://localhost:9080/ACFWeb/SubscriberServlet?request=<request_type>
   <request type> can take one of the following values:
   1. get
   2. add
   3. remove
2. The status of the web service request is displayed

Appendix D – Test Data

Stock Trading Scenario

- Stock Exchange Authority Service schema
  - Service Schema S:
    company: {ticker, price, change}

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:j.0="http://www.cs.rit.edu/~akr1005/">
  <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/change">
   <j.0:IS>CLASS</j.0:IS>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/ticker">
   <j.0:IS>CLASS</j.0:IS>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/price">
   <j.0:IS>CLASS</j.0:IS>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company">
   <j.0:HAS>
    <j.0:company>
     <j.0:IS>CLASS</j.0:IS>
    </j.0:company>
   </j.0:HAS>
   <j.0:HAS rdf:resource="http://www.cs.rit.edu/~akr1005/company/price"/>
   <j.0:HAS rdf:resource="http://www.cs.rit.edu/~akr1005/company/change"/>
   <j.0:HAS rdf:resource="http://www.cs.rit.edu/~akr1005/company/ticker"/>
  </rdf:Description>
</rdf:RDF>
```

- Stock Trader Application schema
  - Application Schema S1:
    Company: {ticker, price, *trading_charges*, change}

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:j.0="http://www.cs.rit.edu/~akr1005/">
 <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/change">
  <j.0:IS>CLASS</j.0:IS>
 </rdf:Description>
 <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/ticker">
  <j.0:IS>CLASS</j.0:IS>
 </rdf:Description>
 <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/price">
  <j.0:IS>CLASS</j.0:IS>
 </rdf:Description>
 <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company">
  <j.0:HAS>
   <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/trading">
    <j.0:IS>CLASS</j.0:IS>
   </rdf:Description>
  </j.0:HAS>
  <j.0:HAS rdf:resource="http://www.cs.rit.edu/~akr1005/company/price"/>
  <j.0:HAS rdf:resource="http://www.cs.rit.edu/~akr1005/company/change"/>
  <j.0:HAS>
   <j.0:company>
    <j.0:IS>CLASS</j.0:IS>
   </j.0:company>
  </j.0:HAS>
  <j.0:HAS rdf:resource="http://www.cs.rit.edu/~akr1005/company/ticker"/>
 </rdf:Description>
</rdf:RDF>
```

- Stock Trader Application schema
  - Application Schema S2:
    company: {ticker, price, change}

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:j.0="http://www.cs.rit.edu/~akr1005/">
  <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/change">
   <j.0:IS>CLASS</j.0:IS>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/ticker">
   <j.0:IS>CLASS</j.0:IS>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company/price">
   <j.0:IS>CLASS</j.0:IS>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.cs.rit.edu/~akr1005/company">
   <j.0:HAS>
    <j.0:company>
     <j.0:IS>CLASS</j.0:IS>
    </j.0:company>
   </j.0:HAS>
   <j.0:HAS rdf:resource="http://www.cs.rit.edu/~akr1005/company/price"/>
   <j.0:HAS rdf:resource="http://www.cs.rit.edu/~akr1005/company/change"/>
   <j.0:HAS rdf:resource="http://www.cs.rit.edu/~akr1005/company/ticker"/>
  </rdf:Description>
</rdf:RDF>
```

References

[1] Aslan, G., & McLeod, D. (n.d.). Semantic heterogeneity resolution in federated databases by metadata implantation and stepwise evolution. *The VLDB Journal, 8*(1999), 120-132.

[2] Berners-Lee, T. (2002). *The Semantic Web*. Retrieved May 13, 2006, from W3C World Wide Web Consortium Web site: http://www.w3.org/2002/Talks/04-sweb/Overview.html

[3] Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., & Wilkinson, K. (2004). Jena: Implementing the Semantic Web Recommendations. In . *In Proc. of the 13th Intl. World Wide Web conference* (pp. 74-83). ACM Press.

[4] Cerami, E. (2002). *Web Services Essentials*. Sebastopol, CA : O'Reilly & Associates, Inc.

[5] *A Discussion of the Relationship Between RDF-Schema and UML* . (1998, August 4). Retrieved May 13, 2006, from W3C Note Web site: http://www.w3.org/TR/NOTE-rdf-uml/

[6] Doan, A., Noy, N. F., & Halevy, A. Y. (2004). Introduction to the Special Issue on Semantic Integration. *SIGMOD Record, 33(4)*, 11-13.

[7] *ebXML - Enabling A Global Electronic Market*. (2006). Retrieved May 13, 2006, from OASIS Web site: http://ebxml.org/

[8] Elmasri, R., & Shamkant, N. B. (2003). *Fundamentals of Database Systems* (4th Ed). Addison Wesley.

[9] Gopalan, B. (2003, February). *Implementing the Active Collections Framework, MS Project Report*. Rochester, NY: Department of Computer Science, Rochester Institute of Technology.

[10] Hakimpour, F., & Geppert, A. (2001). Resolving Semantic Heterogeneity in Schema Integration. In *In FOIS '01: Proc. of the Intl. Conference on Formal Ontology in Information Systems* (pp. 297-308). ACM Press.

[11] Halevy, A. Y. (2005, October). Why Your Data Won't Mix. *ACM Queue, 3*(8).

[12] Halevy, A. Y., Ashish, N., Bitton, D., Carey, M., Draper, D., Pollock, J., et al. (2005). . Enterprise Information Integration: Successes, Challenges and Controversies. In *In Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data* (pp. 778-787). ACM Press.

[13] Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley Professional.

[14] Linthicum, D. S. (2003). *Next generation application integration : from simple information to Web services* . Boston, MA : Addison Wesley Professional.

[15] *OpenCyc.org*. (2006, April). Retrieved May 13, 2006, from http://www.opencyc.org/

[16] Park, J., & Ram, S. (2004). Information Systems Interoperability: What lies beneath? . *ACM Transactions on Information Systems*, (22(4)), 595-632.

[17] Pollock, J. (2005, March). Architecture Patterns for the Enterprise Semantic Web. In *In 1st Annual Conference on Semantic Technology*. Symposium conducted at San Francisco.

[18] Powers, S. (2003). *Practical RDF*. Sebastopol, CA : O'Reilly & Associates, Inc., .

[19] Rahm, E., & Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *The VLDB Journal, 10*, 334-350.

[20] Raj, R. K. (1999, Spring). The Active Collection Framework. *ACM SIGAPP Applied Computing Review: Special Issue on Distributed Computing, 7*(1), 9-13.

[21] Raj, R. K. (2003). Experiences with the Active Collections Framework. In *In Intl. Symp. on Distributed Objects and Applications (DOA 2003)* (Vol. 2519, pp. 1504-1520). Springer-Verlag, LCNS.

[22] *RDF Semantics* . (2004, February 10). Retrieved May 13, 2006, from W3C Recommendation Web site: http://www.w3.org/TR/rdf-mt/

[23] *RDF/XML Syntax Specification (Revised)*. (2004, February 10). Retrieved May 13, 2006, from W3C Recommendation Web site: http://www.w3.org/TR/rdf-syntax-grammar/

[24] *Resource Description Framework (RDF):Concepts and Abstract Syntax*. (2004, February 10). Retrieved January, 2005, from W3C Recommendation Web site: http://www.w3.org/TR/rdf-concepts/

[25] Rosenthal, A., Seligman, L., & Renner, S. (2004). From Semantic Integration to Semantics Management: Case Studies and A Way Forward. *SIGMOD Record, 33*(4), 44-50.

[26] Schlegel, S. (2003, October 1). Implementation Issues in the ebXML CPA formation process the Referencing Problem. In *In Proc. of the 2003 Post Graduate Electrical Engineering and Computing Symposium (PEECS'03)*. Symposium conducted at Curtin University, Perth.

[27] Sheth, A. P., & Larson, J. (1990, September). Federated Database Systems for Managing Distributed, Heterogenous, and Autonomous Databases. *ACM Computing Surveys, 22*(3), 183-236.

[28] *SOAP Version 1.2 Part 1: Messaging Framework*. (2003, June 24). Retrieved May 13, 2006, from W3C Recommendation Web site: http://www.w3.org/TR/ soap12-part1/

[29] Stumptner, M., Schrefl, M., & Grossmann, G. (2004). On the Road to Behavior-Based Integration. In *In CRPIT '31: Proc. of the 1st Asian-Pacific conference on Conceptual Modelling* (pp. 15-22). Australian Computer Society.

[30] Uschold, M., & Gruninger, M. (2004). Ontologies and Semantics for Seamless Connectivity. *SIGMOD Record, 33*(4), 58-64.

[31] *WebServices - SOAP*. (2003, October 15). Retrieved May 13, 2006, from The Apache Software Foundation Web site: http://ws.apache.org/soap/