A Model-Based Framework for Developing and Deploying Data Aggregation Services

Ramakrishna Soma[1], Amol Bakshi[2], V.K.Prasanna[2], Will Da Sie[3]

[1] Dept of Computer Science, USC, Los Angeles, CA
[2] Dept of Electrical Engineering, USC, Los Angeles, CA
{rsoma, amol, prasanna}@usc.edu
[3] Chevron Corporation, San Ramon, CA
Will.DaSie@chevron.com

Abstract. Data aggregation services compose, transform, and analyze data from a variety of sources such as simulators, real-time sensor feeds, etc. This paper proposes a methodology for accelerating the development and deployment of data aggregation modules in a service-oriented architecture. Our framework allows existing semantic web-service techniques to be embedded into a programming language thereby leveraging ease of use and flexibility enabled by the former with the expressiveness and tool support of the latter. In our framework data aggregations are written as regular Java programs where the data inputs to the aggregations are specified as predicates over a rich ontology. Our middleware matches these data specifications to the appropriate web-service, automatically invokes it, and performs the required data serialization-deserialization. Finally the data aggregation program is deployed as yet another web-service. Thus, our programming framework hides the complexity of web-service development from the end-user. We discuss the design and implementation of the framework based on open standards, and using state-of-art tools.

1. Introduction

Data aggregation refers to the process of transforming, composing and analyzing data to produce information that is useful for decision making. Data aggregation workflows are especially important in an enterprise setting where data has to be acquired and aggregated from a variety of heterogeneous sources. Two major problems need to be addressed while creating data aggregation modules. The first problem of *how* the data is accessed is caused by the large diversity of interfaces and protocols presented by the data sources. The second problem of *what* data is being accessed is caused because the data produced by each source has its own syntax and semantics. Our goal is to address these two problems in order to allow domain experts with basic programming skills to easily create data aggregation components that can be plugged into a larger information management architecture.

The work described in this paper is a part of a larger effort on Integrated Asset Management (IAM) for smart oilfields [16]. We envision a framework that will provide a domain expert (in our case, a petroleum engineer or asset manager) simplified access to any piece of data, analysis, and functionality required for use in

making a decision. To achieve this, we have made two key design choices: *service oriented architectures* (SOA) and *model based integration*. Service oriented architecture (SOA) is a style of architecting software systems by packaging functionalities as services that can be invoked by any service requester. Web services are becoming the *de facto* mechanism used for implementing service-oriented architectures and are also employed in our IAM framework. A model based integration framework is built around a shared set of formal models which define the *syntax* and *semantics* of the domain and data elements. All applications in the framework subscribe to these models by producing and consuming data which are compliant with these models. These two technologies in tandem enable data aggregation and application integration in general by standardizing the *how* and *what* of the data accessed.

In this paper we discuss how the problem of data aggregation is addressed in our IAM framework. We assume an enterprise setting where all sources of data are accessible as web-services. We also assume that the schemas of all the data produced and exchanged are known and agreed upon. This is a reasonable assumption in the light of the major thrust on standardization of XML-based data exchange schemas and APIs in the petroleum industry [10]. We build upon these assumptions to create a programming framework which accelerates authoring of data aggregation workflows. There are three main contributions of this paper:

- We present a novel framework for easing the development and deployment of data aggregation applications. The framework demonstrates the integration of techniques from the semantic web for service discovery and invocation into a programming language to provide a rich programming framework for the user.
- We describe the definition of a set of implementation-independent models and show how they are used for automatic service discovery and invocation.
- We discuss a prototype implementation of the framework built on open standards, and currently available tools.

The rest of the paper is organized as follows. Section 2 outlines a simple scenario for data aggregation, which will be the motivating example for the discussion in this paper. In Section 3, we survey existing methodologies and tools that are relevant to our problem and also discuss their shortcomings. An outline of our approach is provided in Section 4, followed by a description of a core set of models and how they are used for service discovery. Section 5 describes the implementation of our prototype service composition and deployment framework. We conclude in Section 6.

2. Motivating Example

To motivate our technique we examine a simple data aggregation workflow from the petroleum industry. Before we explain the workflow itself, we introduce the following terminology from the domain. A *well* is an entity that produces oil, water, and gas. A *block* is a set of wells. The production of a block is the sum of the production of its constituent wells. The oil, water, or gas production for a well or a block is often represented by a “recovery curve” or a “decline curve” for that well or block. A decline curve is a plot of production volume versus time. Decline curves can

also be plotted to show production volume versus the fraction of total oil in place recovered till that time step.

One of the inputs to our aggregation workflow is a data-structure called `WellProductionData`. This data structure holds the production information for a well and the data is produced by a simulator. However, higher-level tools require the aggregated production data at the *Block* level. We will refer to the aggregated data as `BlockProductionData`. To obtain the `BlockProductionData` from the `WellProductionData`, an additional data structure called `BlockData` is used. This data structure contains the information about a block, which contains information like what is the set of wells contained in a given block and some other pieces of data used for aggregation. It is generated either from a database or from another workflow i.e. is retrieved from another web-service. A simplified flow chart showing the workflow `BlockProductionData` from `WellProductionData` is shown below. Note that well-to-block aggregation is not a straightforward summation, and involves some complex calculations like finding moving averages.

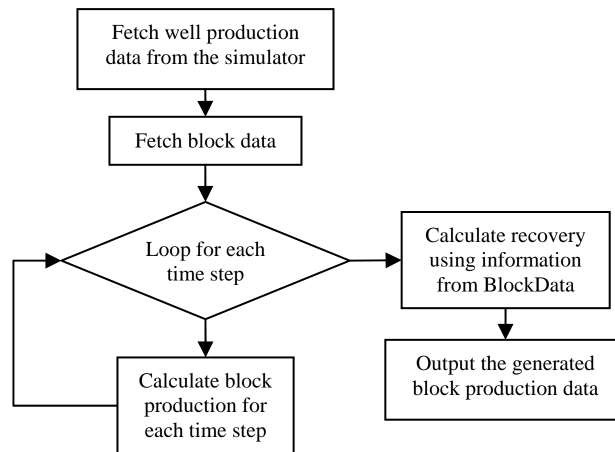

Figure 1. Flowchart for the data aggregation workflow

We want the developer of this aggregation program to be unconcerned with the issues in web service development. These include service discovery, service invocation, and data serialization and de-serialization. The developer could also want the data aggregation program to be deployed into the IAM framework, so that it can be re-used in other aggregation workflows. Thus we require a framework that enables us to create and deploy shared data aggregation workflows without the complexity of understanding web-service standards and deployment issues.

3. State-of-the-art techniques and their limitations

WS-BPEL [1] is the W3C standard for writing web-service compositions. BPEL allows us to script composed services and exposes them as yet another web-service. However, BPEL does not provide support to add complex computations within the

composition - a key feature required to support aggregations. BPELJ[2] is a specification that addressed this shortcoming to some extent. It allowed *snippets* of java code to be embedded within the BPEL script. The main goal was to enable fairly simple calculations and transformations to be embedded within the compositions. It is not clear whether complex computations (such as integrating and invoking a third party tool) can be used. Creating BPEL scripts requires the user to have a good understanding of web-services specifications as well as the where the web-services are deployed. This conflicts with one of our key requirements.

SSIS [3] is a tool integrated with MSSQL server which allows the user to build such aggregation workflows. It consists of a visual composer, where data from various sources including web-services can be retrieved and aggregated. It also allows the user to specify complex transformations and aggregations in a .NET supported language. Although, the user need not write much web-service specific code, he still needs to be aware of the deployed services. The other problem with SSIS is that it is tightly integrated with the rest of the toolkit (.NET, SQL server) and thus integrating with it is not straight forward.

Much work in recent years has been performed on automatic discovery and composition web-services by providing semantic description of the constituent services [4][8]. However, typical semantic web-service composition frameworks like [5][7][11] do not seem to address the general class of applications where the composed service could also include complex computations and transformations. We have used the techniques to describe web-services to define facilitate discovery and invocation. Our system is built for a more *controlled* setting of an enterprise rather than the internet and we confine ourselves to applications that produce data rather those that also change the "state of the world". These differences have helped us to define focused domain models which help us to tailor the semantic-web techniques for our requirements.

Our system is also similar to web-service based *data integration* systems like Prometheus [9]. These systems typically provide a unified database abstraction to a set of web-services and address the problem of how a query is resolved to calls to appropriate web-services. However it is not clear if data aggregations can be defined in these frameworks. Compared to these approaches, we make a simplifying assumption that the produced by each source is a whole relation and thus do not concern ourselves with issues like view integration etc.

Data aggregation and similar workflows occur commonly in scientific workflows. Specialized frameworks like Kepler[17] and Chimera[18] (with concomitant service composition languages) have been employed for implementing such workflows. The major difference between our framework and the above mentioned frameworks is that they make the assumption that the modules containing the aggregation logic already exist and (only) provide methods to "wire" them together (using their special language). In our framework, the aggregation and wiring logic are both developed as part of a single program. This is consistent with one of the primary goals of our system, to avoid the need for the user to learn a new formalism.

Programming languages provide the right level of expressiveness and support we require to create aggregated services. Although the support for creating and consuming web-services in these platforms is becoming more and more seamless, the user still needs to have a good understanding of web-services, platforms and related

issues like XML serialization, SOAP messaging /stub generation etc to be able author such services. Moreover, these languages do not address the problems of intuitive addressing and discovery of web-services. Our technique addresses some of these concerns by providing the user with an abstract data centric interface for writing aggregations. The hard tasks of discovering, invoking web-services are performed by our framework. In essence, our framework marries the expressiveness of programming languages with the semantic web-services idea of using service metadata to facilitate service discovery for authoring web-service compositions.

4 Our Approach

4.1 Overview

Consider a simplified version of the aggregation program from Section 2.

```
1.  /**
2.  * @iam.service
3.  * @author Jane Smith
4.  *
5.  */
6. public class RecoveryCurveAggregator {
7.     /**
8.      * @iam.Operation
9.      * @iam.ServiceMetaData hasInput="blockName" hasOutput="RecoveryCurve"
10.      * @iam.SourceMetaData Producer="Aggregator"
11.      *                     hasRange="Block IN Block_A"
12.      * @return
13.      */
14.
15.     public Reccurve getRecoveryCurve(Object blockName)
16.     {
17.         try
18.         {
19.             DataFactory df = new DataFactory();
20.             WellProductionData csd = df.getData("WellProductionData",
                             "Block=Block_A && Producer="Simulation"", "date>1/1/2005 &&
                                    date < 12/31/2006",null);
21.             BlockWellMap bwm = df.getData("BlockWellMap",
                            "Block=Block_A  Producer=Ontology", null, null);
22.             //Do the aggregation and return the aggregated object
23.             return(this.getReccurve(csd, bwn));
24.         }catch(Exception ex){
25.             System.out.println(ex.getMessage());
26.             ex.printStackTrace();
27.         }
28.             return null;
29.     }
30. }
```

Figure 2. Code snippet for the example aggregation program

The most interesting part of this code snippet is Lines 19-21, which contains the code to obtain the data required for aggregation. The requests for the data are specified as calls the `DataFactory` which abstracts all the complexity of service

discovery, service invocation and data serialization-deserialization. The parameters passed to the `DataFactory` are the type of the object required and a set of predicates that the data is required to have. The user in turn obtains a Java bean which is used to perform the aggregations.

The other important parts of this code snippet are the lines 1-5 and 7-13. These lines contain annotations with in the source code which specify the metadata used while deploying the aggregation program into our framework. The contents of the metadata descriptions are similar to those of a data request. It contains the type of object returned by the service and some predicates describing it.

Once this program is written, it is compiled, debugged as a normal program would be and deployed into our framework. The aggregation program becomes accessible as a web-service and can be similarly searched and invoked in other aggregation programs.

4.2 Modeling

At the heart of our framework is a set of three models which form the basis for a vocabulary for specifying data requests, service advertisements and matching services. These models are described below.

1. Domain model: This is an ontology that models the domain i.e. the entities in an oil-field and their inter-relationships. This model provides an intuitive query vocabulary for the user. For example a user may specify that the data required is the `WellProductionData` for all the *Wells* in a *Block* called Block_A. The notions of what a *Well, Block* and their relationship as a containment is defined in the domain model. A more detailed description of the domain model is out of the scope of this paper and is described elsewhere [15]. A simplified version which only considers parent-children relationships among entities of a domain model is currently used and is defined below.

A Domain object is a 4 tuple Do=<K, N_d, C, P> where:

- K is the kind/class of the domain object (for simplicity we assume here that it is a string),
- N_d is the name of the domain object,
- C = {Do_1,Do_2,…Do_n} is a set of domain objects that are contained by Do,
- P is the parent of the domain object.

A domain Configuration (or scenario) is a 2-tuple, DC=<Na, **D** > where

- Na is the name given to the configuration,
- **D** = {Do_1, Do_2,…Do_n} is the set of domain objects in the configuration.

A domain configuration is the context under which relationships between domain objects are defined. Thus a relationship between domain objects must also specify the configuration under which the relationship will be resolved (e.g. all *Wells* in *Config1.Block_A*).

2. Data Model: The data model is defined as an ontology of classes where each class is defined by a set of meta-data properties and a data schema. The meta-data properties in our system are similar to [13][14] and contains information to identify objects, track their audit trails etc. The schema of a class defines the various properties of the data object. Although the meta-data properties and the data schema are very similar (they define key-value pairs) and the difference between data and metadata is often tenuous, this distinction is very important to us because the metadata is defined and manipulated within our system where as the data schema is defined, stored and manipulated by "external" systems. This approach also allows us to reuse the data schemas published by standard bodies, software vendors etc.

More formally, a class is defined by a 4-tuple C = <N, R, M, S> where,

- N is the name of the class,
- R is the set of parent classes R= $\{C_1, C_2 \ldots C_k\}$,
- M is the set of metadata properties $\{p_{m1}, p_{m2}, \ldots p_{mn}\}$ where P_{mi} is given by a 2-tuple $<T, N_p>$ where T is the type of the parameter and N_p is the name of the property.
- S is the schema for the class (which defines its properties). S is given by a 2-tuple $<N_s, P>$ where N_s is the name of the schema and P is the set of properties given as 2-tuple $<T, N_p>$.

A special kind of class is called the *opaque* class where the class schema has only one property- its value i.e. P = {value}. This kind is used to represent binary and other legacy objects (like ASCII files) in the system.

An object is given by a 4-tuple $O = < C, V_{md}, V_d, \pi >$ where

- C is the class the object belongs to,
- $V_{md} = \{v_{m1}, v_{m2} \ldots v_{mn}\}$ is the set of values assigned to the meta-data fields where each v_i is a 2-tuple $<N_p, V_p>$, N_p is the name of the meta-data property and V_p is the value attribute,
- $V_d = \{v_{d1}, v_{d2} \ldots v_{dk}\}$ is the set of values attached to fields from the schema,
- $\pi = \{v\pi_1, v\pi_2 \ldots v\pi_w\}$ each $v\pi_i$ is a 2-tuple $<N_p, V_p>$; each $v\pi_i$ represents the set of parameters needed to create the object O.

Our framework models both *real* objects and *virtual* objects. Virtual objects are created by services and (in general) have $\pi \neq \varnothing$; on the other hand real objects already exist in some persistent store and have $\pi = \varnothing$. Thus to identify or create virtual objects we also need to specify the input parameters for the service i.e. π.

Finally, we define an ontology as a 3-tuple <C, D, O> where,

- C={ $DC_1, DC_2 \ldots DC_m$ } is a set of domain configurations
- D={$Do_1, Do_2 \ldots Do_n$} is a set of data objects
- $O: S \rightarrow bool$ is a function that takes a statement and informs whether that statement is entailed in the ontology or not.

3. Web-Service Model: The web-service model captures the semantics of the services- so that they can be advertised, discovered and invoked. Since all the web-services in our framework are data providers, the captured semantics of the web-

services are all related to the data. In particular they describe the data provided by the web-service in terms of our data and domain ontology.

A service advertisement in our framework consists of the type of the output the service delivers, types of the input parameters, meta-data predicates that describes the output provided and the range of the data provided by the service defined as predicates over the fields of the output type. A predicate is given by a 3 tuple <P, op, val> where, P is the property asserted upon, op is a operation and val is the value. The expression values, defines a (possibly infinite) set of objects defining the range of objects catered by the service. It can be defined as expressions over primitive data types (int, float, Date etc) or objects/fields from the data and domain models. For the meta-data predicates, the *field* is from the data-model described above, while for the data predicates it is obtained from the schema where the output type is defined.

A service profile is given by a 4-tuple $S = < C_{out}^S, S, MP^S, DP^S >$ where,

- C_{out}^S is the type of the output of the service,
- $S = \{p_1, p_2 \ldots p_k\}$, where each p_i is a 2-tuple $<N_i, T_i>$ represents the input parameters of the service,
- MP^S is the set of predicates over the meta-data $\{mp_1, mp_2, \ldots, mp_m\}$.
- DP^S is the set of predicates over the fields from the schema of C_{out}^S which defines the range of data objects served by the service $\{dp_1, dp_2, \ldots, dp_m\}$.

4.3 Automatic service discovery

Service discovery in our framework involves matching service advertisements (profile) with requests. Please recall that the service advertisements are given by a 4-tuple $S = < C_{out}^S, \pi^S, MP^S, DP^S >$. Similarly, a request is a 4-tuple given by $R = <C_{out}^R, \pi^R, mp^R, dp^R>$. Note that a request tuple is quite similar to an object tuple; because the request identifies a set of objects needed for the aggregation.

Our matching algorithm matches the outputs (C_{out}^S, C_{out}^R) and input parameters (π^S, π^R) as described in [11]. The match is rated as *exact* ($C_{out}^S = C_{out}^R$) > *plugin* ($C_{out}^S \supset C_{out}^R$) > *subsumed* ($C_{out}^S \subset C_{out}^R$) > *fail.*

In addition to inputs and outputs, we also need to match the service and the request predicates. To define the predicate matching, we define a function:

$$INT: P_f \rightarrow D, D \subseteq range(f)$$

Intuitively, INT is an interpreter function that maps a predicate P over a field f to a set of values D. The set D is a subset of all the permissible values of f. We then say that a predicate P_f^S satisfies P_f^R iff $INT(P_f^R) \subseteq INT(P_f^S)$. This is written as $P_f^S \Rightarrow P_f^R$.

While matching the predicates of advertisements with those of a request, three cases occur:

1. *Perfect match*: $P_f^S \Rightarrow P_f^R$
2. *Failed match*: $\neg (P_f^S \Rightarrow P_f^R)$
3. *Indeterminate*: This occurs when for a predicate in the request P_f^R, the service does not advertise a predicate (P_f^S) over the same field. We make an open world assumption and consider an indeterminate match as a potential candidate.

Obviously the scoring function is ordered *perfect* > *indeterminate* > *fail*. All services with even one *fail* predicate match are discarded from being considered as

possible candidates. This is intuitive because, if the user wants data from a Sensor (mp^R:*Producer*="Sensor"), it is not acceptable for her to get data from a Simulator (mp^S:*Producer*="Simulator"), even if it is for the same entity, and with the same timestamp. Thus a service is a match for a given request, if its outputs and inputs are *compatible* i.e. have an *exact* or *plugin* or *subsume* relations and the (metadata and data) predicates all match *perfect*ly or *indeterminate*ly.

Although the INT function is a good abstraction to define the notion of predicate satisfiability, from a more practical standpoint, it is checked by translating $P_f^S \Rightarrow P_f^R$ to an equivalent statement S that can be answered by the (oracle function O of the) ontology.

5 Implementation

The domain and data model are implemented using OWL in our prototype implementation. We have used the OWL-S [4] ontology to represent web-service descriptions. A OWL-S service description consists of three parts: the *service profile* which describes *what* a service does and is used for advertising and discovering services; the *service model* which gives a description of *how* the service works and the *grounding* which provides details on how to access a service. The OWL-S standard prescribes one particular realization of each of these descriptions, but also allows application specific service descriptions when the default ontologies are not sufficient. We have used this to define our own ontologies to describe web-services.

a. Service Profile: This ontology contains the vocabulary to describe service advertisements. We store the information described in the Web-service model here. As recommended in the OWL-S spec, we store these predicates as string literals in the owl description. In the next section, we describe how these advertisements are matched up with user specifications.

b. Service Model: The service model ontology describes the details of how a service works and typically includes information regarding the semantic content of requests, the pre-conditions for the service and the effects of the service.

In many data-producing services it is common that the parameters for the service actually define predicates over the data-type. For example a service that returns `WellProductionData` may contain parameters `startDate` and `endDate` that define the starting and ending dates for which the data is returned. We capture the semantics of such parameters as predicates over the fields. Thus the parameter `startDate` can be defined by the predicate "`WellProductionData.ProductionDate < startDate`". By doing this, we alleviate the need for the user to learn all the parameters of a service and rather let the user define the queries as predicates over the data object fields. Please note that not all parameters can be described as predicates over the data fields. For example, a fuzzy logic algorithm producing a report may require a parameter which describes the probability distribution function used. This parameter has no relation to the data being produced and is modeled here.

The default service model defined in the OWL-S standard, also defines the semantics of input parameters using the *parameterType* property which points to a specification of the class. Currently we do not model the pre-conditions and effects of the services. We do not model the effects of the service because of our assumption

that the services are data producing services and do not change the state of the world. We do acknowledge that pre-conditions may be required in many cases and we intend to address this as part of our future work.

c. Service Grounding: The service grounding part of a model describes protocol and message formats, serialization, transport and addressing. We have used the *WsdlAtomicProcessGrounding* as described in the specification to store the service access related/WSDL data. This class stores information related to the WSDL document and other information that is required to invoke the web-services like the mapping of message parts in the WSDL document to the OWL-S input parameters.

Service advertisements are stored in a UDDI repository in our framework as described in [12]. Please recall from section 4.1 that a data request is programmed as a call to the `DataFactory`. A request in the program is handled by first inferring the closure of data-types that are compatible with the required data. This is used to query the UDDI store to retrieve the service profiles of all compatible services. These are matched according to the ranking criteria scheme described in the previous section and the best candidate is chosen. Currently our predicates can have one of the following operators $\{<, >, <>, \in\}$ and values over basic data-types (Number types, String, Date) and domain objects. Predicate matching for basic types is quite trivial. For predicates involving domain objects the only operator allowed is $\in$, used to define the range of the data sources as (all or some) of the children of a domain object. For example the range of the objects served by a data source can be defined by assigning a meta-data field "`domainObject =` 'Well IN Config1.Block_A'". A query requesting for a data for Well_X i.e. with a predicate "`domainObject =` Well_X" can be resolved by querying the domain model. Please note that once the user has written the aggregations, she can debug it as a normal program. We think that during that process the request can be refined and the web-services are bound as she intended.

Once the candidate web-services are found, the information from the *Profile* and the *Grounding* part of the web-service model is used to construct a call to the appropriate web-service. Please note that some of the parameters are explicitly specified while some of them are a part of the predicates. The parameters which map to these predicates are constructed using the information in the *Profile*. To improve the performance of the system, we store all the Profile information and the Grounding information in the UDDI itself. Thus all required information can be retrieved with one call to the UDDI. We also "remember" the binding associated with a query, thus not incurring the overhead of a UDDI-access. When a call to the system fails (service may be un-deployed or re-deployed), the query is re-sent and the new service information is obtained.

5.1 Service Deployment

After the aggregation program is written it is deployed into a web-service engine. For a typical web-services engine like Apache Axis2[19], this involves creating deployment description document(s), packaging the classes and the dependant libraries as a jar file and copying it into in a specified directory. Apart from this, the OWL-S model for the aggregated service needs to be constructed and saved into a UDDI store. Most of the semantic information describing the service is provided as annotations in the source code by the author of the service. This style of embedding

deployment specific information into the source code is a widely accepted technique in the Java programming community. So, after the author writes an aggregation service as plain java code with annotations, it is deployed by executing a pre-defined Ant script, which creates the deployment descriptors for the web-service engine as well as the OWL-S description documents. In the future we envisage a system with multiple web-service engines where service deployment additionally involves choosing the "best" server. For example an important factor to consider is to minimize the amount of data that needs to be moved. Thus it may be best to deploy an aggregated service on the same machine as (or one "nearest" to) the data producer.

A high-level view of the various elements of our framework and their relationships is summarized in the UML class diagram below[1].

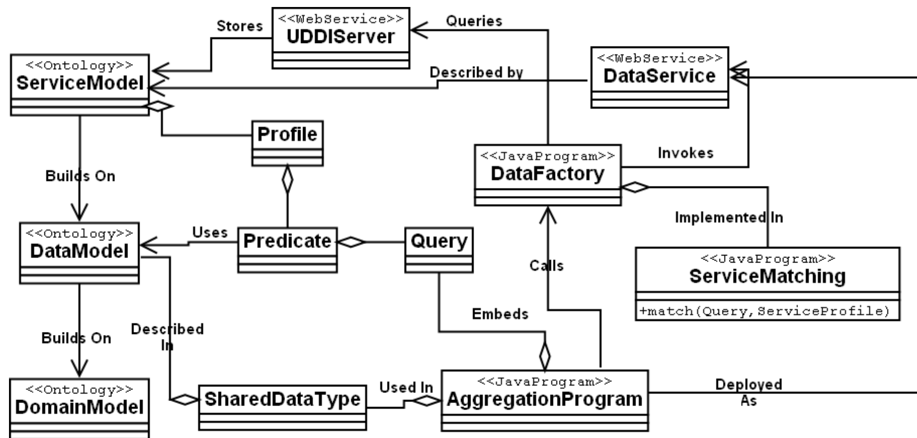

Fig. 3. Major elements of our framework and their interrelationships

6 Conclusions and Summary

In this paper we have presented a framework which eases the development and deployment of data aggregation workflows. The framework integrates techniques for automatic web-service discovery and invocation from the semantic web-services community into ordinary programming languages. This enables the user to write complex workflows in the programming language while using high-level, web-service agnostic specifications to gather the data required for the aggregations. The key element of our framework is the three models in our system: the domain model which captures the semantics of the domain, the data model which captures the data-types and a rich set of meta-data associated with these data types and a service model which captures the semantics of the web services. The data and domain models form the vocabulary for defining the data required in the aggregations, as well as for advertising and matching the web services in the framework. We discussed how the OWL-S standard can be used to hold the information required for automatic discovery

[1] The elements represented as classes in the diagram are not necessarily implemented as java classes- they are more coarse grained modules/data objects.

and invocation. Deployment of aggregation programs in our framework is aided by the use of meta-data embedded within the source code.

7 References

[1] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, IBM, Microsoft, SAP, Siebel, 05 May 2003.

[2] Michael Blow, Yaron Goland, Matthias Kloppmann, Frank Leymann, Gerhard Pfau, Dieter Roller, and Michael Rowley. BPELJ: BPEL for Java. Whitepaper, BEA and IBM, 2004.

[3] Microsoft SSIS: http://msdn.microsoft.com/sql/bi/integration/

[4] D. Martin, et al. OWL-S: Semantic markup for web services. http://www.ai.sri.com/daml/services/owl-s/1.2/overview/

[5] Kaarthik Sivashanmugam, John A. Miller, Amit P. Sheth, and Kunal Verma, Framework for Semantic Web Process Composition, International Journal of Electronic Commerce, Volume 9, Number 2, Winter 2004-05, pp. 71.

[6] http://uddi.org

[7] Daniel J. Mandell and Sheila A. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. Proceedings of the Second International Semantic Web Conference (ISWC2003), pp 227-241, Sanibel Island, Florida, 2003.

[8] J. Rao and X. Su. A Survey of Automated Web Service Composition Methods. In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004, San Diego, California, USA, July 6th, 2004.

[9] Snehal Thakkar, Jose Luis Ambite, and Craig A. Knoblock. Composing, optimizing, and executing plans for bioinformatics web services, VLDB Journal, Special Issue on Data Management, Analysis and Mining for Life Sciences, Vol. 14, No. 3, pp.330--353, Sep 2005.

[10] POSC, The Petrotechnical Open Standards Consortium, http://www.posc.org/

[11] K Sycara, M Paolucci, A Ankolekar, N Srinivasan. Automated Discovery, Interaction and Composition of Semantic Web Services, Journal of Web Semantics, 2003

[12] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Importing the semantic web in UDDI. In Proceedings of E-Services and the Semantic Web Workshop, 2002

[13] Ewa Deelman, Gurmeet Singh, Malcolm P. Atkinson, Ann Chervenak, Neil P. Chue Hong, Carl Kesselman, Sonal Patil, Laura Pearlman, Mei-i Su. Grid-Based Metadata Services, 16th International Conference on Scientific and Statistical Database Management (SSDBM04), June 2004.

[14] Jun Zhao, Chris Wroe, Carole Goble, Robert Stevens, Dennis Quan and Mark Greenwood. Using Semantic Web Technologies for Representing e-Science Provenance In Proceedings of Third International Semantic Web Conference (ISWC2004), Hiroshima, Japan, November 2004. pp. 92-106, Springer-Verlag LNCS

[15] Cong Zhang, Viktor Prasanna, Abdollah Orangi, Will Da Sie, Aditya Kwatra, Modeling methodology for application development in petroleum industry, IEEE International Conference on Information Reuse and Integration, Las Vegas, 2005.

[16] CiSoft IAM project, http://indus.usc.edu/cisoft-iam/

[17] S. Bowers and B. Ludascher. Actor-oriented design of scientific workflows. In 24th Intl. Conf. on Conceptual Modeling (ER), 2005.

[18] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying and Automating Data Derivation. In 14th Conference on Scientific and Statistical Database Management, Edinburgh, Scotland, July 2002.

[19] Apache Axis2: http://ws.apache.org/axis2/