



SPE 99983

A Service Oriented Data Composition Architecture for Integrated Asset Management

Ramakrishna Soma¹, Amol Bakshi¹, Abdollah Orangi¹, Viktor K. Prasanna¹, and Will Da Sie²

¹University of Southern California, Los Angeles, CA 90089 USA

²Chevron Corporation, San Ramon, CA 94583 USA

Copyright 2006, Society of Petroleum Engineers

This paper was prepared for presentation at the 2006 SPE Intelligent Energy Conference and Exhibition held in Amsterdam, The Netherlands, 11–13 April 2006.

This paper was selected for presentation by an SPE Program Committee following review of information contained in an abstract submitted by the author(s). Contents of the paper, as presented, have not been reviewed by the Society of Petroleum Engineers and are subject to correction by the author(s). The material, as presented, does not necessarily reflect any position of the Society of Petroleum Engineers, its officers, or members. Papers presented at SPE meetings are subject to publication review by Editorial Committees of the Society of Petroleum Engineers. Electronic reproduction, distribution, or storage of any part of this paper for commercial purposes without the written consent of the Society of Petroleum Engineers is prohibited. Permission to reproduce in print is restricted to an abstract of not more than 300 words; illustrations may not be copied. The abstract must contain conspicuous acknowledgment of where and by whom the paper was presented. Write Librarian, SPE, P.O. Box 833836, Richardson, TX 75083-3836, U.S.A., fax 01-972-952-9435.

1. Introduction

The goal of an Integrated Asset Management (IAM) framework for the oil and gas industry is twofold. From the end users' perspective, it should offer a single, easy-to-use user interface for specifying and executing a variety of workflows from reservoir simulations to economic evaluation. The framework should not require the user to be an expert in any of the underlying software applications; in fact, the details of selecting, configuring, and invoking the underlying software modules should be hidden from the end user. From the software development perspective, the IAM framework should facilitate seamless interaction of diverse and independently developed applications that accomplish various sub-tasks in the overall workflow. For instance, it should be possible to pipe the output of a reservoir simulator running on one machine to a forecasting and optimization toolkit running on another and in turn piping its output to a third piece of software that can convert the information into a set of reports in a specified format.

Model-based design. The design of our prototype IAM framework is based on the concept of model-integrated system design. The central idea is to define a domain-specific modeling language for structured specification of all relevant information about the particular asset being modeled. The model captures information about many physical and non-physical aspects of the asset and stores it in a model database. The model database is in a canonical format that can be accessed by any of the tools in the IAM framework through well-defined application programming interfaces (APIs). In a

model-based IAM framework, the asset model acts as the central co-ordinator of information access and data transformation. Instead of coupling the various tools to each other through “expensive” pair-wise interface adaptors, each tool is interfaced with the model database. The database thereby enables indirect coupling of disparate applications by allowing them to collaboratively work together in the common context of the asset model. The front-end modeling environment provided to the end user allows definition and modification of the asset model, and also contains a mechanism to allow the invocation of one or more integrated tools that act on different parts of the asset model. A more detailed description of this approach appears in [15] and its application to an integrated forecasting and optimization workflow is described in [20].

Service-oriented architectures. Service oriented architecture (SOA) is a style of architecting software systems by packaging functionalities as services that can be invoked by any service requester. An SOA typically implies a loose coupling between modules. Wrapping a well-defined service invocation interface around a functional module hides the details of the module implementation from other service requesters. This enables software reuse and also means that changes to a module's implementation are localized and do not affect other modules as long as the service interface is unchanged.

Web services form an attractive basis for implementing service-oriented architectures for distributed systems. Web services rely on open, platform-independent protocols and standards, and allow software modules to make themselves accessible over the internet. Web services and service-oriented architectures are becoming a popular and useful means of leveraging Internet technologies to improve business processes in the oil and gas industry [6].

When the service-oriented approach is adopted for designing an IAM framework, every component, regardless of its functionality, resource requirements, language of implementation, etc., provides a well-defined service interface that can be used by any other component in the framework. The service abstraction provides a uniform way to mask a variety of underlying data sources (real-time production data,

historical data, model parameters, reports, etc.) and functionalities (simulators, optimizers, sensors, actuators, etc.). Workflows can be composed by coupling service interfaces in the desired order. The workflow specification can be through a graphical or textual front end and the actual service calls can be generated automatically. Many service composition tools provide such functionality (e.g., [9]).

Data composition. A typical IAM framework will incorporate a number of information consumers such as simulation tools, optimizers, databases, real-time control systems for in situ sensing and actuation, and also human engineers and analysts. The data sources in the system are equally diverse, ranging from real-time measurements from temperature, flow, pressure, and vibration sensors on physical assets such as oil pipelines to more abstract data such as simulation results, maintenance schedules of oilfield equipment, market prices, etc. One of the key components of an IAM framework is an efficient, scalable, and flexible mechanism for collection, aggregation, and *delivery of data in the right format to the right consumer at the right time*. Automating data flow between multiple information consumers will greatly expedite many workflows by eliminating the typically laborious tasks involved in manual preparation of data for input to various tools.

If the service interfaces of different applications are compatible, i.e., if the output of one service can be provided unchanged to another, such coupling is relatively easy. In many workflows, however, intermediate processing is required for the data produced by one tool (service) before providing it to another tool (service). This conversion could be a simple reformatting of data or more complex transformations including unit conversions (e.g., barrels to cubic meters), aggregation (well production to block production), etc. Specific interpolation policies could be required to fill in a data set with missing values. We use **data composition** to refer to this general process of applying a variety of intermediate transformations to data as it flows from one service to another as part of a larger workflow.

In this paper, we present a service-oriented software architecture for data composition in a model-based IAM framework. We discuss the graphical modeling front-end, the data composition language, and the functionality of the IAM compiler that orchestrates the underlying workflow execution based on the users' specification. The design of the software architecture is influenced by the learnings from applying the model-based design methodology to the oil production forecasting use case [20]. A prototype of the data composition framework has been implemented, and the design and evaluation of this data composition framework for a real world use case is planned.

Section 2 presents our domain-specific workflow modeling language with a simple illustrative example. Section 3 discusses the architecture of the data composition framework. Section 4 is an overview of related work, and we conclude in Section 5.

2. Visual modeling of data composition

This section uses a highly simplified real-time reservoir management workflow to illustrate the use of our visual modeling language. In this workflow, a catalog of type curves is available from a series of *a priori* reservoir simulation runs. The curves in the catalog correspond to a set of differing models of the reservoir. As real world production data from the reservoir becomes available, it is to be periodically compared to the type curves in the catalog to estimate the best fit. The type curve(s) that best matches the production data at a given time could then be used as input to other disjunct workflows such as oil production forecasting.

We now analyze this workflow from a data composition perspective and identify the following components:

- *Data sources.* The production data and the recovery curve catalog are the sources of 'raw' data that could be stored in a standard database. Access to the database could be through a web service that provides a query interface for data retrieval and update.
- *Aggregation service.* A software module aggregates time-based raw data (from production as well as simulation), and generates type curves along the desired dimensions - e.g., cumulative oil production vs. reservoir pressure.
- *Pattern matching service.* This software module accepts a set of reference curves from the catalog and a type curve derived from the production data, and performs pattern matching to estimate the best fit.

We now discuss the prototype domain-specific visual modeling language for data composition in the IAM workflow. We used the GME graphical modeling toolsuite [17] to automatically generate a graphical modeling environment from our modeling language specification. Details of the GME tool are omitted here.

Our modeling language consists of three different modeling paradigms. The first (*DataElement*) defines the basic data types that are exchanged between services, the second (*Composition*) specifies the transformations to be applied to the data, and the third (*Domain Model*) links the data composition model to the asset model.

Data schema

The data schema defines the entities and relationships to capture the data types and the methods/transformations on them. Thus the main elements of the data schema are *DataElement* and *Transformation*. A *DataElement* is either a *DataObject*, which is an abstraction of a domain specific object or a *DataPrimitive*. *DataPrimitives* are primitive data types like integer, Boolean, etc.

The other important kind in the data schema, *Transformation* is used to define transformations on *DataElements*. A *Transformation* can either be an *ObjectTransformation* which is a predefined transformation on the *DataObject* entities or *CustomTransformation* which refers to user-defined transformations. Each *Transformation* has an associated attribute called *Formula* which specifies the data processing

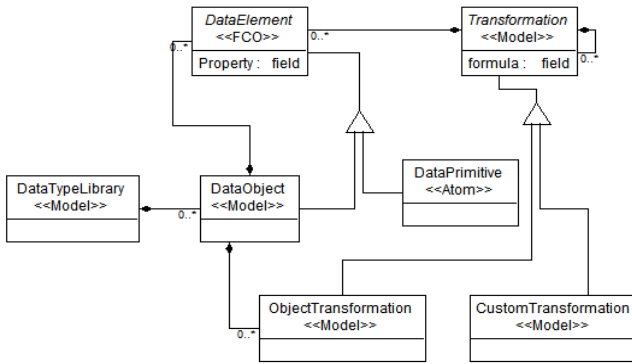


Figure 1: The data schema

that needs to be done in the transformation. Currently, the formula is a block of text that specifies a sub-routine in a standard programming language such as C.

The first step in using our framework is to construct a library of the identified *DataObject* types and *Transformations* (or *methods* in object-oriented terminology). These objects are then instantiated by the user while composing a specific workflow.

Data composition schema

This schema (Figure 2) defines the entities that are required to compose workflows using the elements from the data schema. The main kind in this schema is *Composition*. A *Composition* contains *DataElements* and *Transformations*. The type of the *DataElements* used in the compositions is obtained from the library of *DataObjects* described above.

While specifying data composition, it is not sufficient to indicate the types of data to be transformed. In addition, it is necessary to specify which instances of that type of data are to be ‘composed’. For example, a composition might only use data related to a particular reservoir volume element (block). We accomplish this by allowing the user to define the *range* of the data to be used, in terms of elements from the particular asset model. This specification is done in a separate aspect of the model, called the *Properties* aspect, where the user provides a declarative expression to define the conditions that the data he requires needs to satisfy.

Although there is an overlap between the elements in the data schema and the composition schema, the reason for separating them is to clearly distinguish the data definition aspect from the data composition aspect. The data definition stage, where the domain objects are identified and defined (ideally) occurs just once. These objects are then used many times just as a library is used in a programming language in the composition stage.

Constants to be used in the data composition can be declared by setting the *isConstant* property of the *DataItem* to true.

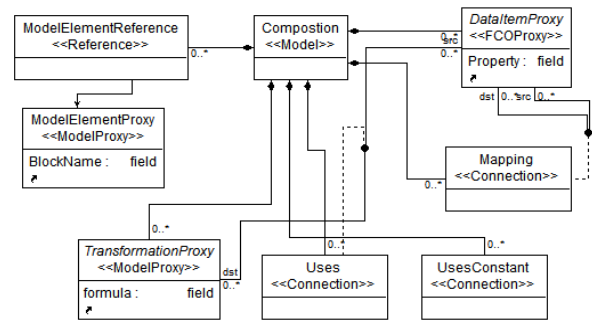


Figure 2: The data composition schema

Finally, to make a composition reusable, we have provided each composition with input and output ‘ports’. Thus, a user-defined composition model can be reused in other workflows in the same way as the built-in *Transformation* object. A *Mapping* connection exposes the data produced by a composition as ports so that the composition can be reused.

It is important to note that our modeling language is totally independent of notions of web services, etc., although the concepts of web services and SOA are the key enablers of our frameworks. Instead, the focus of the modeling language is on specifying the data objects and transformations, without worrying about how the data is sourced and where the transformations are carried out.

Domain model schema

A domain modeling paradigm is used to specify the asset. Each element in the model (representing a physical or non-physical aspect of the asset) has data associated with it, which represents some relevant information like the current state/configuration of the asset. The main goal of the domain model schema is provide mechanisms to keep this information updated, by using the results from a data composition workflow to update the suitable section of the asset model. The domain model schema lets the user specify the elements of the model database to be updated by the results of the composition.

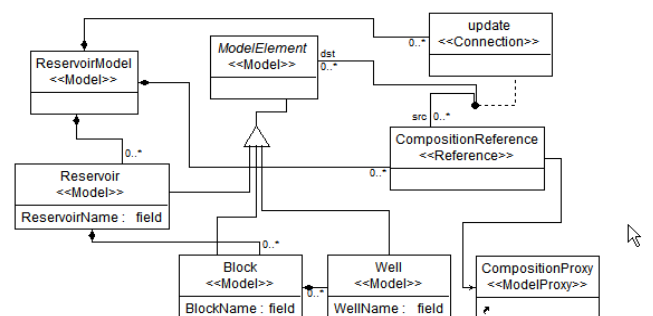


Figure 3: The domain model schema

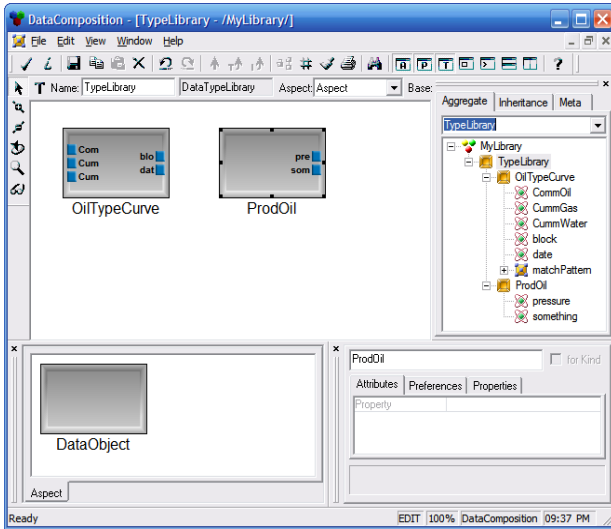


Figure 4: Modeling the data type library

The domain model schema presented in Figure 3 is a small and highly simplified schema for modeling a reservoir. In this model, *Reservoirs*, *Blocks* and *Wells* can be represented. The *update* element allows the user to specify that the results of the composition can be used to update the model database. A more comprehensive model is described in [20].

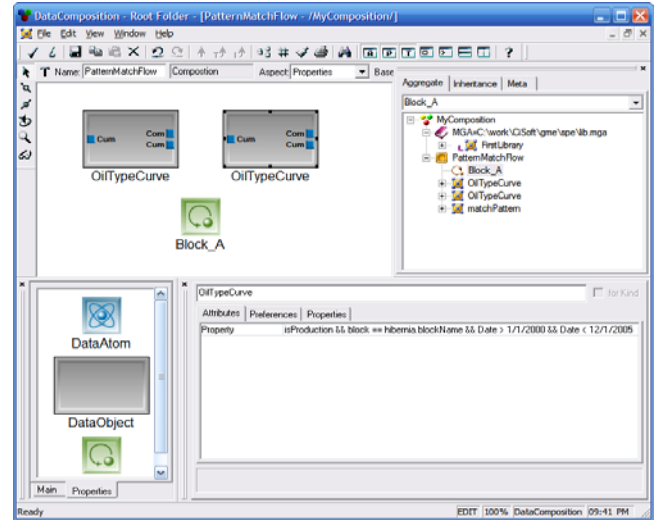
Illustrative example.

To illustrate how the language is used, we will use the problem of the type curve matching described previously.

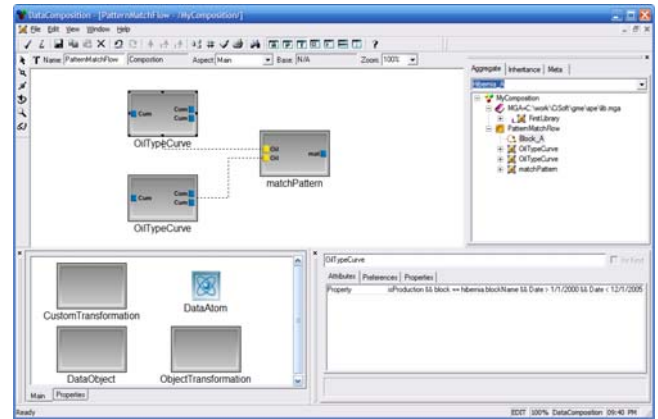
First the data objects are defined in a type library as shown in Figure 4. The figure shows a simple data type library that contains a few data-types including *OilTypeCurve*. This type curve object is an abstraction used to represent a schema that includes cumulative oil production, cumulative water production etc. It also encapsulates a transformation called *matchPattern* which compares two oil type curves and returns a similarity index.

To describe the composition, we create a project based on the *Composition* schema. The type library defined previously is imported into the project, and provides the building blocks for the composition model. A new *Composition* object is instantiated, and two *OilTypeCurve* objects are added to it.

Next, the properties of the objects are described, for example, to specify that the type curve is required for the block named *Block_A*. This is done in the *Properties* aspect as shown in Figure 5(b). The other properties are also defined declaratively on the data objects. The property field of the two *OilTypeCurves* is shown in in Table 1. Note that the “*Block_A*” in the property specification is a reference (pointer) to the *Block_A* object in the composition model. Thus, the context of the specification forms the namespace for resolving the references in the properties declaration. The



(a)



(b)

Figure 5. Data composition for pattern matching:
(a) Properties and (b) Main aspects

Block_A object in the composition model in turn links to the corresponding block entity in the asset model.

After this description is presented to the system, it is compiled and the data satisfying the composition is fetched. The details of how this is accomplished, is the focus of the next section.

Property a:

```
src="simulation" && block= Block_A.blockName && Date > 1/1/2000
&& Date < 12/1/2005
```

Property b:

```
src="production" && block = Block_A.blockName && Date > 1/1/2000
&& Date < 12/1/2005
```

Table 1: Specifying the source of data in the composition model

3. System Architecture

The architecture is based on two goals: *generality* and *reuse*. Generality means that our approach is applicable to

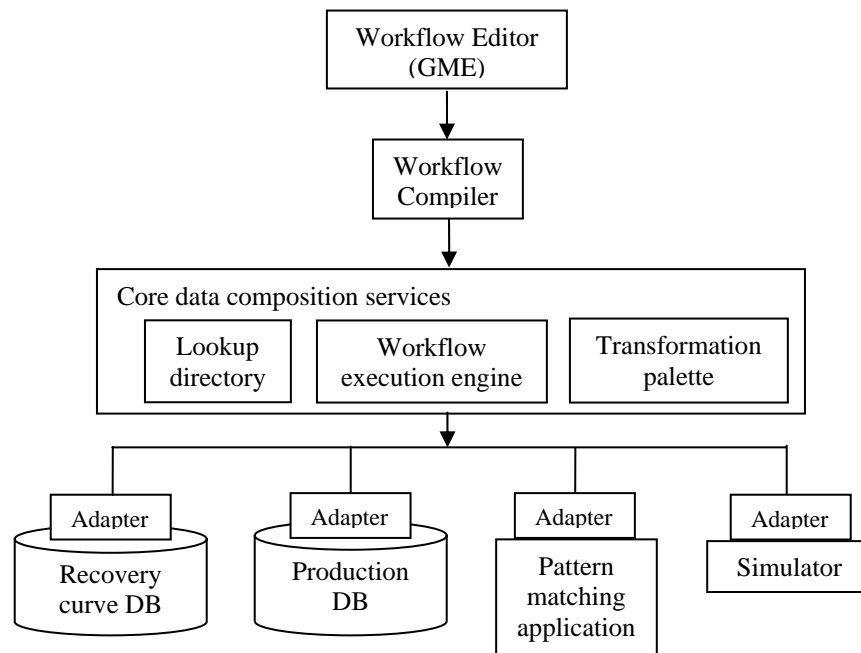


Figure 6. Overview of our system architecture.

many different data composition scenarios. Reuse means that architecture use as many off the shelf components as possible.

A language based approach described in the previous section, helps us achieve generality because, the range of applications supported by the framework is dependent on the expressiveness of the language and is quite large. However, a big challenge of this approach that is described below is how the “programs” in the language will be executed.

As mentioned earlier, the main goal of our system is to enable the decision maker (reservoir engineer) to describe the workflow that produces the data she requires to make decisions. Workflows in our systems are defined in terms of domain objects, a set of pre-determined “methods” of those objects and a set of workflow primitives, exactly like in a conventional object oriented language. These descriptions are then compiled by the IAM-COOL compiler to produce a workflow consisting of a series of service invocations. The output of the compiler is a schedule that can be executed by a workflow engine like MS SSIS [14] or a BPEL [8] engine. The compiler makes use of a lookup directory, which keeps a mapping of the service that caters a specific data, much like the UDDI yellow pages. Additionally, the lookup directory can keep track of other metrics like data quality to choose the best data source when multiple data sources serve the same data. The information required for the lookup directory is provided by the data-sources themselves. Legacy data sources are provided with wrappers to enable this. The data sources (wrappers) provide the required meta-data including the type(s) of data they provide, data quality indicators etc, which is indexed by the lookup directory. The compiled workflow is

then given off to a workflow engine for execution. Finally, a service called the transformation palette provides a set of pre-defined transformations that can be used in the workflow. The high-level architecture diagram is shown in Figure 6.

3.1 Lookup component

The lookup component is a key component in our architecture and is used to translate the high level object descriptions to corresponding service invocations. To achieve this, the lookup needs to keep track of the data and transformations provided by the data or computational source. It does this by storing meta-data for each service. In particular, the lookup component keeps the following meta-data for each data source:

1. **Source Metadata:** This metadata is used when the requester knows the source from which the data needs to be fetched. The source metadata can also provide hints about the quality of the data supplied by the data source. Dublin core metadata schema is commonly used to define the source meta-data [16].
2. **Type of objects** served by the source: Is the key information that allows the directory to resolve the specifications to the data sources.
3. **Range of objects** served by the source: A data source may supply only a certain range of the data objects. For example oil production data for reservoirs in a particular block of the reservoir. Please note here that we assume that a data source always serves all the fields of the data type.

4. **Transformations on data objects** performed by the service. This meta-data contains a mapping of the object method to the corresponding port of the service providing that method.
5. **Data quality meta-data:** Many indicators of data quality have been identified in literature including freshness/recency of the data, completeness of the data, accuracy of the data etc. This information is used when more than one data source supply the same piece of information and the system needs to choose the right piece of data for the decision to be made.

To increase the scalability of the system, the lookup directory can be implemented in a distributed fashion much like the DNS [18]. In such a system, the lookup component is not a single monolithic component but is rather composed of a multiple components organized hierarchically, with each lookup component in the hierarchy indexing a subset of the data sources. When the “root” lookup component receives a request for some dat/transformation, it delegates the request to the right component in the hierarchy.

3.2. Compiler

The IAM-COOL compiler takes the high-level description from the user and converts it into an executable workflow like BPEL. One of the main tasks in this process is to translate the high-level object references to calls to the actual data-sources serving that data. The compiler does this by requesting the lookup directory to provide the best data-source for the required data type and the quality metrics. The compiler then uses this to produce a BPEL schedule that contains the sequence of web-service calls that need to be performed. The compiler also converts the custom transformations specified in the description to appropriate calls to the transformation palette component.

3.3. Services

Both data and computational resources are abstracted as web services in our system. This abstraction provides us with uniform interface and protocols to address each resource, considerably decreasing the complexity of integration. Apart from providing the data and computational resources, the web services in the system provide the meta-data information to the framework. In general each service has the following interface:

```
IAMCOOLService{
  Init();
  Stop();
  XMLDoc getData (String objType, Query spec);

  //Set of data transformations it provides.
  XMLDoc transformation1();
}
```

Table 2: The interface of a data-source

Init is the initialization process where the data sources advertise themselves to the lookup directory and provide it with the meta-data described above. The stop method is called when the service needs to be shutdown. This method is the inverse of the init method where the directory removes the current service as providing the data and the transformations that it advertised in the init process. In the getData method of the interface, the data source finds the data that is of the same type as the first parameter and matches the data specification. It returns a XML document containing the required data. We plan to use Xquery [5] as the language for specifying the queries. This approach gives us a powerful and well-defined query language to specify our constraints. Also, Xquery is XML based, an open standard, well-understood, and has many supporting tools. The simplest (but inefficient) way of using Xquery would be to first generate the XML document and then apply the Xquery constraints on it to obtain the subset of the data satisfying the requirements. Techniques to convert Xquery queries to their SQL counterparts have been discussed in literature [3] and can be used for more efficient data retrieval for data residing in SQL supported databases.

In building such systems, most of the data sources already exist (legacy data sources) with their own proprietary interfaces. A well-accepted technique (design pattern) to integrate such legacy data/computational sources is to provide them with **wrappers** [1]. The wrappers provide a web-service abstraction to the data source and present the above-mentioned interface to the system.

3.4. Workflow engine and transformation palette

We plan to use an off-the-shelf workflow execution engine like MS-SSIS [13], BPEL execution engine, etc. to execute the schedules produced by the IAMCOOL compiler. An interesting aspect that has to be considered here is the error handling - what happens if a service in the schedule is down for some reason? An elegant way to do this would be to use the custom extension mechanism of BPEL, to make provision to list more than one service that can satisfy the data requirement. This would also entail that the execution engine be modified to understand these custom extensions and use an alternative data source in the case of failure.

The last component in our architecture is the transformation palette. The transformation palette provides the workflow designer with a set of transformations that can be applied to the data from the services. A simple set of primitives including the relational operators like project, select, join etc., mathematical and aggregation/statistical operators like add, multiply etc. make the framework more powerful.

4. Related work

Our work is related to and draws from many areas of research. One key goal of our work is to propose a Domain Specific Visual Language (DSVL) for the system. Although much work has been done in the area of DSVL [5], we are not aware

of a language that addresses the needs for data acquisition for the petroleum industry. One design choice we could have made was to use an existing workflow language like BPEL [8] or visual workflow languages like JOpera [9], instead of our own DSL. However, this choice was not pursued because most workflow languages are designed with computer/software engineers in mind and work at very low level of abstractions.

Our architecture has been inspired by many systems. The idea of writing “intentional” programs has been drawn from the notion of intentional naming [10] where entities are addressed by the service they provide rather than their physical address. The idea of using a lookup directory to map names to the physical locations is pervasive in all kinds of middleware systems like RPC [11], CORBA [12] etc.

In [2], the authors describe an architectural style and a architecture for a similar “data-intensive” application. Consequently, their architecture resembles ours in many ways. The *profile server* of their architecture is very similar to the lookup directory in our system and the *resource server* is similar to a service/ wrapper of our system. However, in our system we incorporate the notion of data quality and transformations, not addressed in theirs. We also provide a visual language “front end” that allows the domain engineer to fetch and compose data, which is not present in their system.

Cohen et al. [4], describe an architecture for data composition in a system with a huge number of data sources and rapidly changing data (called iQueue). Like our system, the goal of the system is to allow the users to write applications by “focusing on the semantics by facilitating the mechanics” of compositions. Like our system, they also consider data quality metrics while choosing the data source. Their systems includes a component called “data resolver” similar to our lookup directory and composer manager similar to our “compiler”. However our system has a scope narrower that iQueue in terms of the kinds of data that it deals with; we only consider data relevant to petroleum industry. Consequently, we have made use of this fact to propose the notions of a set of data objects and fixed set of operations on them as the basis of describing compositions and a visual specification language. Unlike iQueue, we do not address issues like security and the use of multiple protocols because we assume the presence of a SOA. Finally, unlike in iQueue, one of the important goals of our system has been to incorporate off the shelf components wherever applicable.

Much work in recent times has been done in the area of effective discovery of webservices. Apart from the data that the webservices are mandated to publish as part of WSDL and UDDI standards, various kinds of meta-data have been proposed to help in the discovery process [16]. In [13], the authors describe how ontologies and semantic markup can be used for effective service discovery. Here, the services add semantic information using DAML-S to indicate the semantics of the services they provide. This information is held within the WSDL descriptions and special UDDI structures and is

used for reasoning about and choosing the appropriate services. It is possible to use meta-data as ontological descriptions in our system and we plan to use them in our future versions of our framework. The system we have described has a larger scope than these systems and service discovery is only a part of our system. Another difference is that our system is deployed in a “closed” environment of an organization as opposed to “open” environments of the internet, and thus our problem is simplified to a large extent.

5. Discussion

One of the main problems that our system addresses is non-standard data formats i.e. each data source of the system could provide the data in its own format/schema. Efforts such as the POSC [7] projects, have been initiated in the community today to address this problem. The goal of these efforts is to define a schema or a common vocabulary for transparent data exchange between various tools and systems. XML-based open standards such as the WITSML data schema and API are under active development. Our proposed framework benefits from such standards mainly because in the absence of such industry-wide standards, data exchange without loss of information between different tools becomes difficult. Also, our framework requires the end user to have some familiarity with data schemas and composition templates. By using data schema(s) developed by standards bodies such as POSC, acceptance and eventual deployment of the framework is facilitated.

Acknowledgments

This research was funded by CiSoft (<http://cisoft.usc.edu>) – a joint USC-Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies.

References

- [1] Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources, VLDB 1997
- [2] C. Mattmann, D. Crichton, J.S. Hughes, S. Kelly and P. Ramirez. Software Architecture for Large-scale, Distributed, Data-Intensive Systems. In *Proceedings of the 4th IEEE/IFIP Working Conference on Software Architecture (WICSA-4)*. pp. 255-264. Oslo, Norway, June 2004.
- [3] Ioana Manolescu, Daniela Florescu and Donald Kossmann: "[Answering XML Queries over Heterogeneous Data Sources](#)", Proc. of the Int'l. Conf. on Very Large Databases (VLDB) 2001, Roma, Italy
- [4] Norman Cohen, Apratim Purkayastha, Luke Wong, Danny L. Yeh, iQueue: A Pervasive Data Composition Framework, Proc of the Third International conference on Mobile Data Management
- [5] David S. Wile, Lessons Learned from Real DSL Experiments. In *Proceedings of the 36th Hawaii International Conference on System Sciences*. Kona. Jan 2003.

- [6] R. Gregovic, R. Foreman, D. Forrester, and J. Carroll. A common approach to accessing real-time operations data - Introducing service-oriented architecture to E&P, SPE ATCE 2005.
- [7] Petrotechnical Open Standards Consortium, <http://www.posc.org>
- [8] T. Andrews et al., Business Process Execution Language for Web Service v1. 1
<http://www.ibm.com/developerworks/library/wsbpel>
- [9] Cesare Pautasso, Gustavo Alonso [The JOpera Visual Composition Language](#) Journal of Visual Languages and Computing (JVLC), 16(1-2):119-152, 2005
- [10] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The Design and Implementation of an Intentional Naming System. In Proceedings of the ACM Symposium on Operating Systems Principles
- [11] Andrew D. Birrel and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39-59, February 1984.
- [12] CORBA, <http://www.corba.org>
- [13] K. Sivashanmugam, K. Verma, A. Sheth, J. Miller, "[Adding Semantics to Web Services Standards](#)," The 2003 International Conference on Web Services (ICWS'03), Las Vegas, NV, June 2003, pp. 395-401.
- [14] SQL Server Integration Services (SSIS), <http://msdn.microsoft.com/SQL/bi/integration/default.aspx>
- [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] DCMI, Dublin Core Metadata Element Set, Version 1.1: Reference Description, Dublin Core Metadata Initiative, 1999.
- [17] The Generic Modeling Environment (GME), <http://www.isis.vanderbilt.edu/projects/gme/>
- [18] Berkeley Internet Name Domain (BIND). <http://www.isc.org/bind.html>, 2004.
- [19] David Trastour, Claudio Bartolini, and Javier Gonzalez-Castillo. A semantic web approach to service description for matchmaking services. In Proc. International Semantic Web Working Symposium (SWWS), 2001.
- [20] Cong Zhang, Abdollah Orangi, Amol Bakshi, Will Da Sie, and Viktor K. Prasanna. Model-based framework for oil production forecasting and optimization: A case study in integrated asset management, SPE Intelligent Energy Conference and Exhibition (IECE), April 2006.