Integrating Heterogeneous Data Sources using the COIL Mediator Definition Language

Christian Och, Roger King, Richard Osborne

Database Research Laboratory
Department of Computer Science
University of Colorado
Boulder, Colorado 80309 – 0430, USA

och@cs.colorado.edu

ABSTRACT

Megaprogramming, module interconnection languages, and mediator-based approaches built on standard distributed object technologies such as CORBA and DCOM have provided promising advances in enterprise-level data integration. These distributed object technologies, however, still require relatively low-level, technology-dependent implementations to achieve object (or module) interconnection, communication, and coordination. The Component Object Interconnection Language (COIL) is a language designed specifically to facilitate rapid and flexible data integration through high-level object interconnections. COIL is an extensible language, with both declarative and imperative aspects. It has been designed to provide broad functionality in declaring and controlling structural and semantic data integration, and includes constructs for specifying updates, constraint enforcement, and update propagations. This paper presents an overview of COIL and a stylized example showing an application of the COIL language and its components.

Keywords

Data Source Integration and Evolution, Heterogeneous Databases, Interoperability, Mediators, High-Level Integration Semantics, CORBA

1. INTRODUCTION

Generally, the integration of heterogeneous data sources is performed in an ad hoc manner leaving all the accessing and integration work to the client of these data sources. Therefore, the tremendous burden of semantically interconnecting data sources in a robust fashion is left to the client application. Distributed object technologies such as CORBA [5] and DCOM [6] provide a standardized framework of object interaction at a conceptual/interface level. The CORBA client can depend on a common interface technology (CORBA IDL) and does not have to deal with several different access technologies. However, these distributed object technologies are by themselves not sufficient for the integration of heterogeneous data sources.

Integration is generally approached by generating a wrapper around the data source which provides an object level interface to the client. Within these wrappers, data source specific access technologies are used such as ODBC, JDBC or even RPC's. The client is now able to access data from different data sources using a single technology. The difficult task of integration of that information from diverse interfaces is still left to the client.

Sanctuary addresses this problem by shifting the main workload of the integration process from the client to Sanctuary. Sanctuary (formerly Sybil) is a data source integration environment that works on top of distributed object technologies [7]. Sanctuary was successfully used in several integration processes of different heterogeneous data sources. An example of the integration of data stored in old legacy CODASYL databases and data stored in O2 object oriented databases is presented in [8]. With Sanctuary, integration is shifted from the client to special objects, called mediators, and is performed using a language and environment specialized to the task of performing data integration. Instead of dealing with several interfaces and performing the integration work on the client side, a mediator in the Sanctuary environment provides the user with a single (user-defined) interface and the integration of the data from the heterogeneous data sources is handled by the system.

COIL (Component Object Interconnection Language) is used to model, specify and generate those mediators that exist as first-class objects in the Sanctuary environment. The semantics among data sources, the integration process and the interface provided to the client are defined within the COIL language. COIL allows one to express the semantics of large-scale data integration among highly heterogeneous data sources in a quick and lightweight fashion. The use of distributed object technologies allows data sources to participate as members of a COIL mediator, in addition to keeping its status as an autonomous data source. In essence, distributed object technologies abstract the access technologies from the data source. COIL abstracts the integration process from the client.

In this paper, we introduce and illustrate the COIL language and its components with an example. This example exhibits how COIL resolves different types of heterogeneity in the integration of heterogeneous data sources — namely differences in data representation, global constraints across data sources, and inadequate query facilities in legacy systems. In addition, the example shows how COIL is used to perform set operations on structurally heterogeneous data and the creation of views over data from various data sources (specifically an object-oriented DBMS, a relational DBMS, and a legacy flat file system). It should be noted that the discussion and example presented herein describe only a subset of the features provided by COIL.

2. THE COIL MEDIATOR DEFINITION LANGUAGE

Sanctuary is a data source integration and evolution environment for supporting the construction of large scale systems that integrate heterogeneous data sources. The key component in the Sanctuary system is a special kind of object called a mediator (mediators in the context of information management systems were introduced by Wiederhold in [12]). A Sanctuary mediator is a first class object in some distributed object environment. The semantics of the integration process among the involved data sources, the operations and the data transformations performed on data from the data sources, and the accessing of the data sources is all captured within the mediator object. A more detailed description of Sanctuary mediators and their runtime environment is presented in Section 3.1.

The COIL mediator definition language allows for the definition of those mediators. COIL provides a set of high-level programming constructs called *COIL components* for performing data integration. Those components are used to define the internal architecture of the mediator as well as to specify the semantics of the integration tasks performed by the mediator. The COIL components are used in the mediator specification to define and express the internal architecture and the integration tasks performed by the mediator on a higher semantic level than it could be expressed with a standard programming language. Each COIL component provides an interface by which it can be accessed by the user's chosen standard implementation language (i.e. C++ or Java) allowing a very tight coupling of the integration process performed by the COIL mediator and its components to the standard implementation tasks performed by the user's native programming language. Therefore a COIL mediator definition consists of code specific to the integration performed using COIL components, as well as code written in a standard programming language that handles standard implementation tasks. The COIL components are specifically designed for integrating heterogeneous data sources and are one of the primary strengths of the mediation process in the Sanctuary system. Specifically, the integration facilities provided by the COIL components allow for the resolution of potential integration problems due to structural or semantic heterogeneity. A more detailed description of the COIL components and their functionality is given in Section 4.

In addition to the definition of the integration tasks performed by the mediator, the interface used by clients of the mediator (i.e. client applications or other mediator objects) is defined in the COIL program. It is possible to specify multiple interfaces for the same mediator so that the mediator can be used in different distributed object technologies. In that case the COIL program has to contain a separate export definition for each technology in which the mediator will be used. In a third section of the COIL program all data server objects that are used by the mediator have to be declared. It is possible for the mediator to use data servers that use different CORBA orbs, as well as data server objects that exist in different object technologies (e.g. DCOM). It is, of course, also possible for the mediator to be a client of other Sanctuary mediators. The structure of a COIL program and its different sections involved are explained in more detail in Section 3.

The COIL compiler generates the code for the mediator in the user's native programming language (e.g. C++ or Java) and with special code generated for the specific distributed object environments specified in the COIL program. Therefore the COIL components use the native programming language's data model. The generated code from the COIL compiler can then be integrated into the user's normal development environment to generate the executable mediator object. A mediator generated from a COIL program is a first class object in some distributed object technology: a Sanctuary mediator is an instance of a COIL program.

The mediator object encapsulates the integration tasks and semantics specified in the COIL program. Because a mediator is a first class object it is possible to have multiple instances of the same mediator. Each instances performs the same integration tasks but has its own local state and its own local data and data structures. The compilation process of a mediator generated from a COIL program is depicted in Figure 1 below.

It is important to mention that the output of the COIL compiler contains all the information needed to allow the mediator to interact with objects that use different CORBA orb technologies and even with objects from different object technologies such as DCOM if those are specified in the COIL program. All those implementation details are hidden from the mediator designer through high-level COIL components.

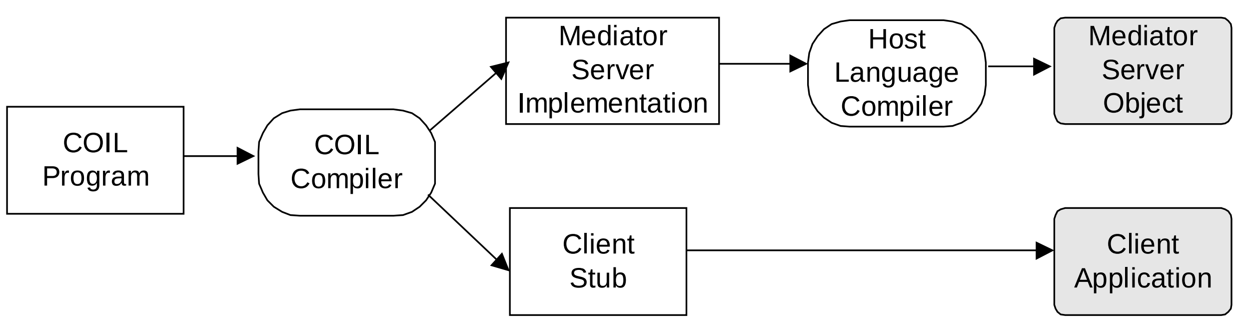

Figure 1 – The COIL Mediator Compilation Process

3. THE GENERAL STRUCTURE OF A COIL MEDIATOR DEFINITION

A COIL mediator definition consists of three parts: the import section, the export section and the mediation section. The mediator presents an interface to its clients. The syntax of this interface depends on the chosen distributed object technology (e.g. for CORBA objects an IDL interface is generated). In the *export* section, the interface exported to the mediator's clients is defined. The *import* section contains the definitions of all objects (from the distributed object environments) used by that mediator.

Note that the *import* section can include objects from different distributed object technologies. Also mediator objects can be used in the *import* section of other mediators, allowing the chaining of mediators.

The semantics of the mediator and its functionality are defined in the mediation section of the mediator definition. The mediation section is subdivided in two subsections: the mediator architecture section and the exported function definition section. The mediator architecture starts with the keyword 'mediator_architecture', followed by COIL definitions that reflect the semantic architecture of the mediator. The COIL definitions in this section include the specification of the COIL components used, their structure, how they interact with each other, and how the COIL components are connected together. Within this section COIL components such as docks, filters, and routes are used to define the architectural framework for the data used and managed within the mediator. Those definitions include the specification of the ways on which the data is shipped within the mediator (i.e. routes and docks), the definition of operations that are performed on data types of different (heterogeneous) types (e.g. set operations and mappings), and the definitions of restrictions that are put on data shipped through the mediator (e.g. filters and transformations).

In the second subsection of the mediation section the semantics for the functions exported by the mediator are presented. Implementation code for every function defined in the export section has to be given using the chosen host language. In the definition of the exported functions, the COIL components defined in the mediator architecture sub-section can be used in addition to the standard host language features.

Note that this section contains only code of the chosen standard programming language. As mentioned above, every COIL component provides an interface to the implementation language by which it can be accessed and manipulated. In the current version, the COIL compiler provides language bindings for Java and C++. The interface of a COIL component can be used in the implementation section to reconfigure the component. Depending on the component, it is for example, possible to activate/deactivate the component or to change specific runtime parameters of the component and therefore change the behavior of the component. Specifically, the reconfiguration of COIL components in the implementation section also affects the internal architecture of the mediator and therefore the semantics of the integration process performed by the mediator. Therefore it is possible to alter the behavior of the mediator and therefore the semantics of the integration process at runtime. The main focus in the definition of the mediator's architecture (and therefore in the integration process performed) is on the following issues:

- the handling, manipulation, and storage of data of structural heterogeneous types from heterogeneous data sources
- the definition of mapping operations that allow the transfer of data between heterogeneous data types and data sources
- the provision of a uniform way to handle data sources of different distributed object technologies
- the definition of routes on which data is shipped within the mediator
- the definition of operations (i.e. set operations), transformations and restrictions (i.e. filters) that are applied to the data shipped on those routes
- the specification of the order in which the operations, mappings, transformations, and filters are applied to the data shipped through the mediator

An example COIL mediator and its runtime environment is depicted in Figure 2 below. The figure shows the COIL mediator 'ABC' and the different data sources participating in the integration process. The different sections of a COIL mediator and other important aspects of COIL and its runtime system are also shown in Figure 2, including:

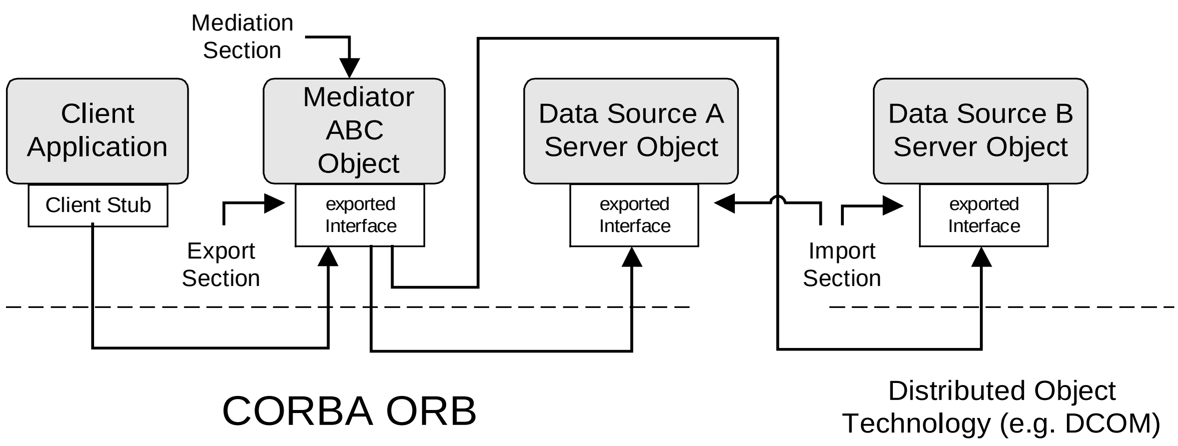

Figure 2 – The Sections of a COIL Mediator Definition

- The client application only interacts with the mediator object not with the data sources. Changes to implementations of the data source servers do not affect the client application. Even changes to interfaces of the data source servers do not affect the client application's implementation code.
- COIL is not restricted to a particular distributed object technology. The data source servers might be implement in different technologies. In Figure 1, data source server A is a CORBA server, whereas data source server B is implemented using DCOM technology.

The *export* section of the COIL program is shown as well: it is represented as the exported interface of the mediator object. The client application acts as a client of the mediator object: it uses the automatically generated client stubs to interact with the mediator object.

A skeleton of a COIL program is presented in Figure 3. The code segment shows the general structure of a COIL mediator definition as well as the different sections of a COIL program. The code represents the skeleton of the code for the example mediator ABC.

```
mediator ABC
{
  export namespace "CORBA"
  {
    // functions provided by the mediator:
    // definition of the mediator's interface
  }

  import namespace "CORBA"
  {
    // data source A interface definition
  }

  import namespace "DCOM"
  {
    // data source B interface definition
  }

  mediation
  {
    mediator_architecture
    {
      // definition of the internal
      // architecture of the mediator:
      // definition of the COIL components
      // and how they are connected together
    }
    // implementation of the functions
    // provided by the mediator using
    // the COIL components defined in the
    // Mediator_Architecture and the chosen
    // standard programming language
}
```

Figure 3 – The Skeleton of the COIL Definition for Mediator ABC

3.1 COIL Mediators

As described above, COIL mediators exploit encoded knowledge about a specific set or subset of data and provide information for a higher level of applications. The information provided by the mediator is retrieved from various underlying data sources. Conflicts due to heterogeneity on different levels (e.g. structural/semantic heterogeneity) is resolved within the mediator so that a homogeneous interface is presented to the mediator's clients. As an extension of the three-layer mediator architecture introduced in [12], a five layer architecture is used for the COIL mediator runtime environment. The five functional layers of the run-time environment of a mediator object are depicted in Figure 4 below.

- The *application layer* is where client objects exist, which correspond to the traditional notion of applications.
- The *broker layer* consists of the collection of brokers, possibly in different environments, that service requests between client objects, mediators, and the objects whose data the mediators integrate.
- The *data integration layer* consists of the community of mediators that perform integrations of data provided by the underlying data server layer.
- The *data server layer* consists of objects that encapsulate data and services provided by the local data sources being integrated.
- The *data source layer* consists of the actual data sources involved in the integration process such as DBMS', information stored in raw files etc.

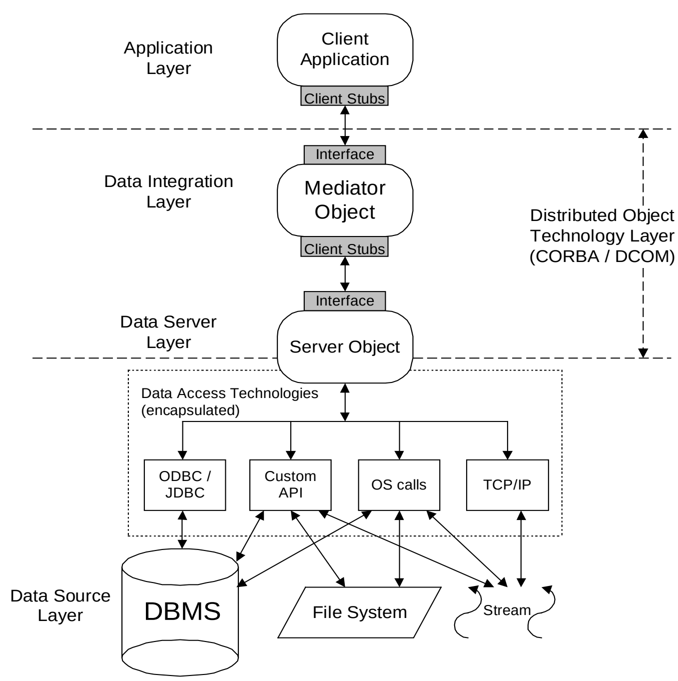

Figure 4 – The COIL Mediator Runtime Environment

3.2 Contributions of COIL

Model and Specify Integration Semantics and the Architecture of a Mediator on a high level of abstraction

Perhaps the most salient contribution of COIL to the literature is the fact that COIL allows for the modeling and specification of the semantics of the integration performed by the mediator and its architecture on a very high level of abstraction. COIL allows one to reason about the semantics of the integration task performed by the means of semi-declarative, high-level components. Usually mediators are hand-coded and all integration tasks are manually performed using a standard programming language. COIL provides high-level constructs to specify the internal architecture of the mediator and therefore to specify the semantics of the integration performed. In addition, COIL provides special data integration operations (such as the set operations and relational operations) to ease the integration process.

It is important to mention that COIL performs those set operations on collections of objects. The objects might be objects from a standard programming language, objects from a distributed object technology or a combination of both. The problem of impedance mismatch does not arise in COIL as it does in other approaches (e.g. OODBMSs) because the COIL set operations are performed directly on objects in the programming language environment (COIL uses the standard programming language's data model). Therefore COIL enriches the standard programming language (or the distributed object environment respectively) with the ability to perform set operations.

The definition of the integration semantics on a high level substantially improves the understanding of the mediator and its integration process. It also aids in the maintenance of the mediator as well as possible reuses of the mediator.

Lightweight Integration Approach – support for evolving environments and requirements

Integration in COIL is performed in a fast and lightweight fashion: no global schemas are generated, no wrappers have to be written. One can completely concentrate on the integration effort at hand. A mediator is used to perform one (or several) integration tasks. The mediator only has to know about the data sources that participate in the integration and might be unaware of other data sources that might be present in the system. Adding new data sources, for example, to an existing integration (i.e. to several mediator objects) is a very easy task because COIL works on top of existing distributed object technologies and no wrappers around the data sources have to be built.

The lightweight integration approach is very suitable for evolving environments and changing requirements. Integrating new data sources, or deleting existing ones, for example, is a fairly easy task. Only the mediators affected by the environment change need be changed, no global data structures or schemas have to be maintained and updated. Using the high-level components provided by COIL, it is a fairly easy task to adjust the affected mediators. Changes in requirements or inter-relationships between data sources can also be handled in a very straightforward way. In addition COIL enables one to perform set operations and relational operations (i.e. the construction of views) on structural heterogeneous information, provides facilities to define global constraints among the data sources participating in the integration, and provides constructs to resolve structural and semantic heterogeneity.

4. COIL LANGUAGE COMPONENTS

COIL provides several special language features called COIL components. The COIL components are used within the mediator architecture section of a mediator specification to model and define the internal architecture of the mediator on a higher semantical level than it could be expressed with standard programming languages. There are different COIL components which are used for the different aspects of the mediator's architecture definition. A COIL component consists of two parts: a compile time definition and a runtime part. In the mediator architecture, the semantical architecture of the mediator is defined using the special COIL components.

In the compile time definition part, every COIL component that is used in the mediator must be defined. The compile time definition includes the definition of the structure of the COIL components, their functionality, and the definition of the data types on which they are working (i.e. mapping operations, transformations etc.). In addition, the routes on which data is shipped through the mediator and how COIL components are grouped together (i.e. set operations, and the order in which COIL components are executed) are defined in the architecture section.

As mentioned before, every COIL component provides a specific interface (API) by which it can be accessed by the chosen standard programming language. The API of a COIL component is the runtime part mentioned above. Depending on the component, different API functions are provided. These functions can be used in the definition of the exported functions in the mediation section of a mediator definition. The API allows the COIL programmer to configure the component at runtime. For example, the API of a COIL filter provides functions that enable the COIL programmer to change the values of several filter parameters, and the API of a COIL route allows for the activation or deactivation of components that are located on the route.

Space does not allow a detailed discussion of the COIL components. A brief overview of the most important components is given below: Some example COIL code that shows how the internal architecture of a mediator is defined using COIL components is presented in Section 5.2. A more detailed discussion of the COIL components, their syntax and API, and how they are used to define the internal architecture of a COIL mediator is given in [9].

Dock: Docks are components that hold data and are the terminal sources (or destinations) of data transshipment within a mediator. Services provided in the dock API include loading and unloading data on and off the dock, as well as functions that describe the state (i.e. number of elements loaded on the dock) and services to ship the data loaded on the dock to data structures that are available to the host language.

Mapping: Mappings are components that are used to perform structural conversions on data. Structural mappings involve field names and data types.

Filter: COIL filters are used to place restrictions on the data shipped between docks. Each filter has a specified criterion by which data elements may pass. The filter's API allows the host language to change specific filter parameters at runtime so that the filter can be configured according to the user's needs.

Transformation: A transformation is used to value mappings of data elements that pass through it. Value mappings provide semantic conversions of data, such as a floating point value conversion from centigrade to Fahrenheit. A transformation's parameters can be changed at runtime to enable the user to write generic transformations for a range of data types. All elements read into a transformation are passed through. The transformation's API is similar to the filter API.

Route: COIL routes are used to define the paths and order of components along which data are to be shipped inside the mediator. A route is defined between two docks and may contain other, intervening COIL components. The order of the components on the route dictates the order in which components are applied to the data that are shipped between its source and destination docks. The runtime API of routes include functions to activate and deactivate specific components on a route. If a component gets deactivated, it is not applied to data that is shipped on the route.

Operation: Two routes with the same destination dock can be combined to an operation. In that case the generic COIL construct 'COIL_Operator' has to be placed on both routes. As part of the COIL operation API, the 'COIL_Operator' can be replaced by an actual operator (e.g. set intersection, or the view operator) at runtime. The operation is performed on the data when the data would be shipped through the 'COIL_Operator' construct.

Comparator: If data of different types (from two different docks) is used in an operation, a comparator which is used to define the equality of two elements from the different docks must be defined. With respect to relational databases, the comparator can be understood as the definition of the key fields in the different data types on which set operations (e.g. joins and unions) are performed. Comparators do not have API's of their own but are used in the COIL operation API to specify which comparator to use for a specific operation.

Operators: COIL operators are not COIL components of their own. They do not provide an API and they do not have to be defined. Similar to comparators, COIL operators are used in the COIL operation API to assign a specific operator to an operation component. COIL provides the following operators: union, intersection, difference, combination and the COIL view operator.

Object Binding: Object bindings are used to provide a uniform way to bind or connect the mediator to specific (possibly named) objects of some distributed object technology. They also allow for the assignment of a name to the object by which it can be accessed within the mediator.

5. THE ONLINE BOOKSTORE EXAMPLE

In this section, the COIL components and concepts are demonstrated in an example in which a mediator definition is presented. The example presents the definition of the mediator 'Online Bookstore' which is, as suggested by the name, used as an interface to a virtual bookstore. The bookstore Emazon.com presents the following functionality (interface) to its client applications:

- Get detailed information for a specific book specified by its ISBN.
- Get books written by a specific author. This information does not include the stock information.
- Search for books that match specific keywords – detailed information about books that match keywords passed by the client are returned to the caller application.
- Get only books for a specific language.
- Order books – a list of books (ISBN) and the quantity of books to order is passed to Emazon. In return the caller retrieves a detailed order list that includes information about the status of the ordered books (ordered/out of stock – reordered from the publisher) and an order number which can be used for further inquiries about the order.

In addition to the interface presented to its clients, the mediator has to perform several other integration tasks to fulfill several internal requirements. If, for example, the client application orders books from Emazon, several constraints have to be checked. If the client orders a bigger quantity of one (or more) of the books than the quantity that Emazon has on stock, the book has to be reordered from the publisher and the customer has to be informed about the reorder.

5.1 The System Architecture of the Online Bookstore

The bookstore is built on top of existing, completely heterogeneous data sources. Due to the different data sources involved in the example, several types of heterogeneity are present in the example and are discussed in detail later in this section. The book information is retrieved from two different servers that provide heterogeneous interfaces. One of the book servers sits atop old legacy data stored in a flat file system. Therefore the query facilities provided by the first book server are very limited and some of the query functionality and its semantics must be provided by the mediator. The second book information server sits on top of an OODBMS and provides a much more powerful interface to its clients than the legacy book server.

Because of the underlying DBMS, the second book server provides richer query facilities. In addition, it is assumed that the information from the DBMS is more up to date and accurate than the information in the legacy files. If duplicate information exists (information about a book is found in both servers), the data from the DBMS server should preferably be used.

The third data source involved in the online bookstore mediator handles the business processes of Emazon. It contains the stock information of the and handles the orders of books by customers, as well as the reorders of books from the publisher if a book runs out of stock. The book warehouse server is implemented using a traditional relational DBMS. This server does not contain any specific data about the books that are available in the book warehouse (such as the book title, the author or the measurements of the book). Retrieving the book data from other data sources saves memory space in the book warehouse server and ensures that the newest information of a book is retrieved without having to worry about updating the book data.

The system architecture of the online bookstore is depicted in Figure 5 below. It shows the involved data sources and the role of the COIL mediator in the integration of the different data sources.

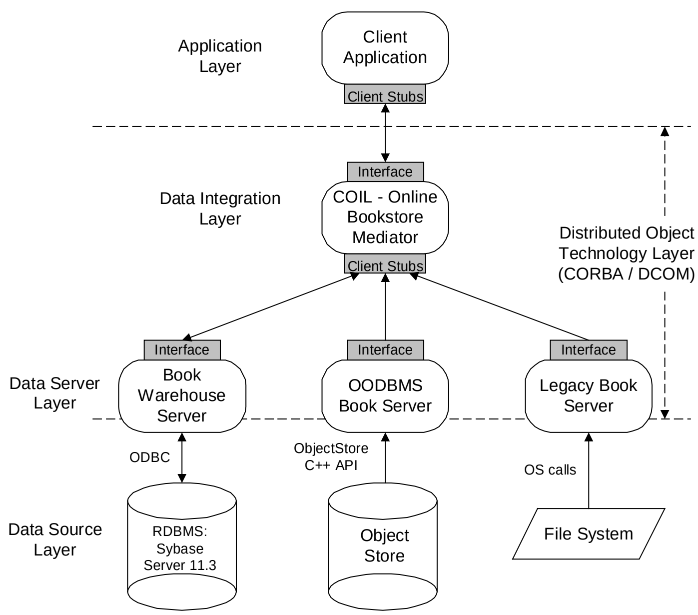

Figure 5 – The System Architecture of the Online Bookstore

5.2 The Architecture and the Semantics of the Online Bookstore Mediator

The internal mediator architecture is depicted in Figure 6 below. The figure displays the COIL components used (and how they are combined) to provide the functionality presented to the clients of the mediator. Within this section the mediators architecture and its internal semantics are described. In addition, it is shown how the COIL components that make up the mediator's architecture are used to solve the heterogeneity problems described above.

As shown in the figure, the three docks 'Books_1', 'Books_2', and 'Cache_Books_2' are used to load data (books in the various representation formats) from the book servers. Depending on the task that has to be performed, the data from two of the docks is combined using a COIL operation and shipped to the dock 'All_Books'. In the example, the 'union' operator is assigned to the named operation 'Book_Union'. Therefore a set union over the data loaded on the two origin docks is performed and the result is loaded on the 'All_Books' dock. The COIL union operator works similar to the union operation in a relational DBMS – duplicate elements are contained only once in the result set. Note that another operator could be assigned to the COIL operation component so that, for example, a set intersection is performed on the data from the two docks.

The COIL operation resolves heterogeneity problems that might occur due to the fact that structural heterogeneous data types from different data sources participate in the operation. The COIL operation component allows one to perform set operations on structural heterogeneous data.

COIL provides a language construct that permits one to specify key fields based on which elements of structural heterogeneous data types can be compared and set operations performed. In our example, the ISBN of the books is defined as the key field based on which books from the different data sources are compared. Although not necessary in our example, the definition of the key fields may contain complex constructs and even user-defined functions.

The book data loaded on dock 'Books_1' is retrieved from the DBMS data source, whereas the book data loaded on the docks 'Books_2' and 'Cache_Books_2' is retrieved from the legacy server. Because the book data type from the DBMS server captures more information, the type of data that can be loaded on the 'All_Books' dock is of the DBMS book type. To perform the union operation and in order to ship data from the 'Books_2' dock to the 'All_Books' dock, a COIL mapping between the different data types has to be defined. The types of heterogeneity arising from the differences in the data representation (name differences, format differences, and structural differences) are resolved by a mapping. The mapping describes how an element loaded on the origin dock is mapped to an element that is shipped to the destination dock. For example, COIL allows for the mapping of the field 'Title' to the field 'Book_Title'.

From every dock that loads data from one of the data sources, a COIL route is defined on which the data is shipped to the 'All_Books' dock. Several COIL filters are placed on the book routes. A language filter is placed on every route. The interface of the mediator enables the clients to set the language filter to a specific language. Only books that are written in that language can pass the language filter. Filters are used to put restrictions on the data shipped between docks and can be seen as enhancements to the query facilities provided by the data sources (which might not be present at all).

Filters can also be used to provide more complex query facilities. Through the mediator, a query can be issued that returns all books that match specific keywords. The legacy data source does not provide a keyword query service. The COIL filter 'Keyword_Query' is used to provide the keyword query facility for the legacy server.

The filter is placed on the route from the 'Cache_Books_2' (which contains a local copy of all books stored in the legacy server) to the 'All_Books' dock. Whenever the client application issues a keyword query, all books from the 'Cache_Books_2' dock are shipped to the 'All_Books' dock,

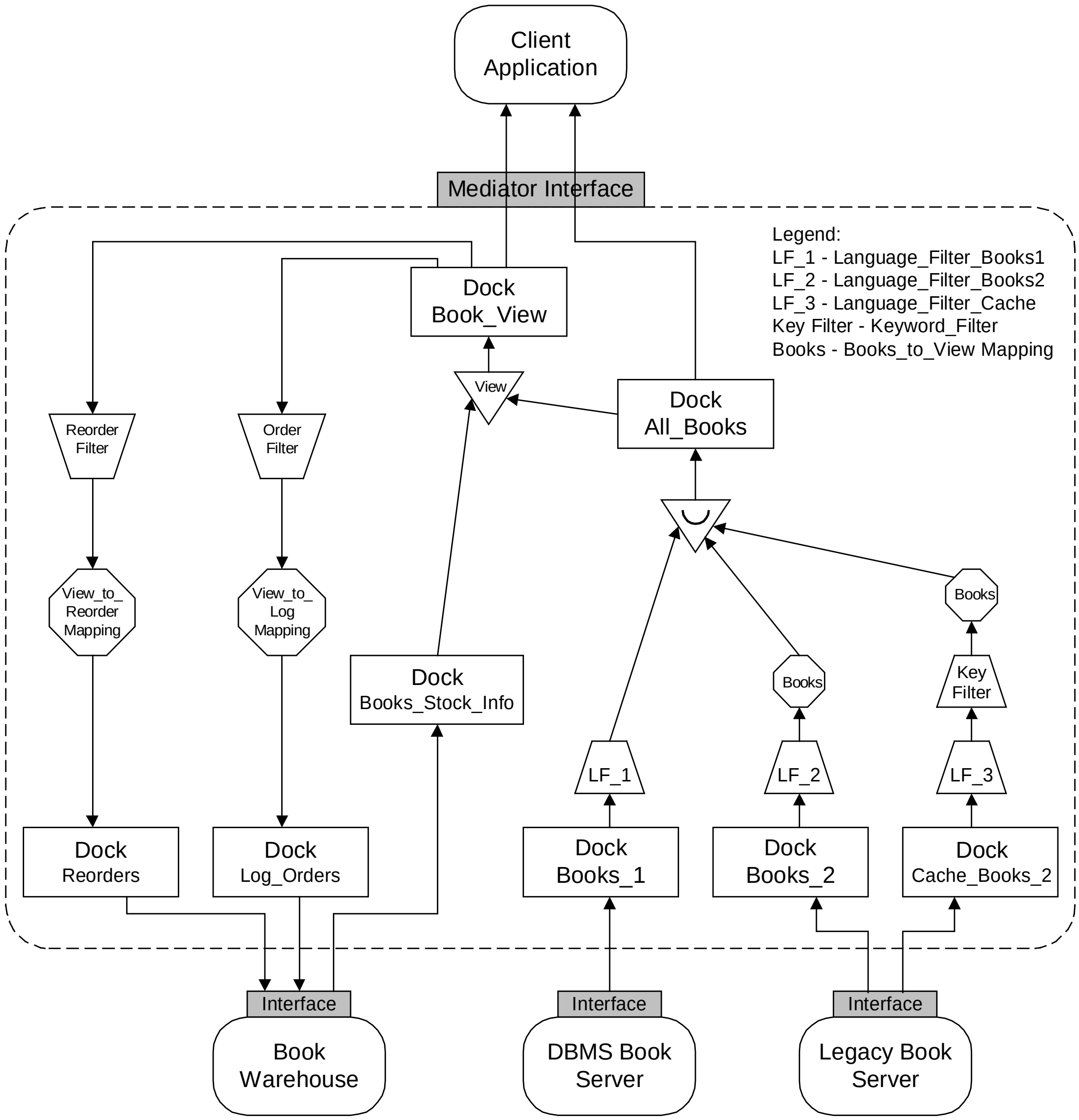

Figure 6 – The Internal Architecture of the Online Bookstore Mediator

but only the books that can pass the keyword filter will reach the ‘All_Books’ dock. COIL provides ways to encode the desired query facilities inside the filter.

Parts of the COIL mediator definition for the bookstore mediator is depicted in Figure 7 below. The presented COIL code segment is part of the ‘mediator_architecture’ section that describes the internal architecture of the mediator. Referring to Figure 6, the code segment describes the right-hand portion (essentially all COIL components that participate in the union operation) of the mediator architecture. The code example shows how the COIL language is used to connect previously defined COIL components together (e.g. the code shows how the ‘Books_1’ dock is connected to the ‘All_Books’ dock). Note that the ‘mediator_architecture’ section consists purely of COIL language constructs. The first part of the section is used to define the COIL components and their semantics (e.g. docks, filter, and mappings). The second part (the part shown in Figure 7) is used to connect those COIL components together to form the internal architecture of the mediator. Because of the limited space, the COIL code for the COIL component definitions could not be shown and other parts of the mediator definition had to be omitted as well. However, a detailed description of the COIL components, their syntax, and semantics is presented in [9].

Another important feature of the COIL language is illustrated by the example and is reflected in Figure 6: the API’s provided by the different COIL components can be used to configure the mediator’s architecture and the mediator’s semantics at runtime. An example for a runtime configuration of the mediator is the reconfiguration of the filters. The COIL filter API, for example, can be used to assign different parameters to the language filters so that books written in different languages can pass the filter.

```
mediator_architecture
{
// COIL component definitions for Mappings and Filter (omitted)
...
  // create the docks used in the mediator
  create Dock All_Books of CORBA::Book_Server::BookInfo;
  create Dock Books_1 of CORBA::Book_Server::BookInfo;
  create Dock Books_2 of CORBA::Legacy_Book_Server::Book;
  create Dock Cache_Books_2 of CORBA::Legacy_Book_Server::Book;

  // connect the docks: create the routes and operations
  // create the union operation that combines
  // the books from the two data sources
  create Route Books_1_Route connect Books_1 to All_Books;
  create Route Books_2_Route connect Books_2 to All_Books;
  create Route Cache_Books_2_Route connect
                             Cache_Books_2 to All_Books;
  insert COIL_Operator on Books_1_Route;
  insert COIL_Operator on Books_2_Route;
  insert COIL_Operator on Cache_Books_2_Route;
  create Operation Book_Union combine Books_1_Route,
                                      Books_2_Route,
                                      Cache_Books_2_Route;

  // place the filters and the mappings on the routes
  insert Books_Mapping on Books_1_Route;
  insert Books_Mapping on Books_2_Route;
  insert Language_Filter_Books1 on Books_1_Route;
  insert Language_Filter_Books2 on Books_2_Route;
  insert Keyword_Query_Books2 on Cache_Books_2_Route;
}
```

Figure 7 – Parts of the COIL Definitions for the Online Bookstore Mediator

The behavior of the keyword filter can be changed similarly: the keywords for which the filter is searching among book titles can be changed at runtime. In addition, the API's of the COIL language components can be used to completely activate and deactivate specific components. If the user, for example, decides to retrieve books written in any language, the language filters will be deactivated. It is also possible to deactivate or activate an entire route.

The semantics behind the usage of the routes from the 'Books_2' dock and the 'Cache_Books_2' dock to the 'All_Books' are an example of the activation and deactivation of an entire route. Only one of the two routes is activated at a specific time.

If a keyword query is issued to the mediator, the 'Cache_Books_2' dock is activated and the 'Books_2' dock is deactivated so that only books loaded on the 'Cache_Books_2' dock participate in the union operation. For all other queries (query for a specific author, a specific ISBN) the 'Cache_Books_2' is deactivated and the data loaded on the 'Books_2' dock is used in the union operation.

5.3 Supporting Views

The dock 'Book_View' is used to provide an integrated view over data from different data sources. Stock information data from the book warehouse server is combined with detailed information about books from the two book servers. The data loaded on the 'Book_View' dock therefore contains general information about a book (e.g. the author, the title, and the ISBN) as well as detailed stock information from the book warehouse server (such as the price of the book and the number of available copies). Data from the different sources is combined into a single data structure. As shown in Figure 6, the COIL 'view' operator is used to generate the integrated view of the data. It is essentially a view operation but with the important difference that the result of the view operation is not stored in another table but in objects that can be accessed by the client. The Pegasus system [14] introduced a similar functionality to generate integrated views over different object-oriented databases. Similar to the COIL view operator, a conversion function that allows for the comparison of key fields from the different databases has to be defined in Pegasus to create the integrated view.

6. RELATED WORK

A number of technologies have been developed for interconnecting large-grained modules and information sources. They provide frameworks for composing, reconfiguring, and reusing systems comprised of wrapped components. Approaches here include Megaprogramming, mediator languages, module interconnection languages, and database interoperation facilities.

Megaprogramming [11] is a technique for programming with software modules called Megamodules that implement a collection of services and facilities provided by large systems or organizations. Module interconnection languages, such as LILEANNA [13], are used for application-level module composition, and for reasoning about changes within those composites.

The idea of using mediator [12] architectures for integrating heterogeneous sources of data is now well established. An array of powerful data integration approaches have been developed within the context of mediator architectures, including [2], [3], [4]. Our work is particularly influenced by TSIMMIS [2] and MIND [3]. TSIMMIS integrates information in a common data model, Object Exchange Model (OEM) [10], which can then be queried by applications. Translators (or wrappers) map data from underlying information sources into the OEM and mediators refine information from translators and embeds knowledge necessary to process that information. The TSIMMIS environment also allows constraint management.

COIL mediators perform the integration process directly atop existing distributed object technologies and therefore no translators have to be written. In addition, COIL uses the data model of the programming environment (e.g. C++) as its underlying data model. Therefore no mappings to a common data model (e.g. the TSIMMIS OEM model) have to be provided.

MIND (METU Interoperable Database) is a CORBA-based multi-database system. Queries in MIND are executed using the public interfaces on objects that encapsulate local database agents. A global database agent encapsulates the global schema and interfaces with the local database agents. MIND assumes that the global database agent and local database agents all use a standard interface. In contrast to MIND, COIL does not require participating data sources to provide a standard interface and therefore no wrappers around the data sources have to be built.

References

[1] O. Bukhres, A. K. Elmagarmid, and E. Kuhn. Advanced Languages for Multidatabase Systems. Chapter in “Object-Oriented Multidatabase Systems”, A.K. Elmagarmid, O. Bukhres (eds), Prentice-Hall. 1996

[2] H. Garcia-Molina , Y. Papakonstantinou , D. Quass , A. Rajaraman , Y. Sagiv , J. Ullman , V. Vassalos, J. Widom . “The TSIMMIS approach to mediation: Data models and Languages.” Journal of Intelligent Information Systems. 1997

[3] A. Dogac, C.Dengi, and M.T. Öszu. “Distributed Object Computing Platforms.” Communications of the ACM. September 1998, Vol. 41. No. 9

[4] Sophie Cluet, Claude Delobel, Jérôme Siméon, Katarzyna Smaga: Your Mediators Need Data Conversion! SIGMOD Conference 1998

[5] The Object Management Group: The Common Object Request Broker: Architecture and Specification, Revision 2.2, http://www.omg.org, February 1998

[6] Microsoft Corporation: Distributed Component Object Model Protocol – DCOM 1.0, http://www.microsoft.com, January 1998

[7] Roger King, Michael Novak, Christian Och, Richard Osborne, Fernando Vélez: The Sybil Database Integration and Evolution Environment: An Overview, ADBIS Symposium 1997

[8] Roger King, Michael Novak, Christian Och, Fernando Vélez: Sybil: Supporting Heterogeneous Database Interoperability with Lightweight Alliance. The Third International Workshop on Next Generation Information Technologies and Systems, NGITS 1997

[9] John Todd, Christian Och, Roger King, Richard M. Osborne, William McIver: “Building Mediators from Components”, International Symposium on Distributed Objects and Applications, DOA 1999

[10] Roy Goldman, Sudarshan Chawathe, Arturo Crespo, Jason McHugh. “A Standard Textual Interface Format for the Object Exchange Model (OEM).” Stanford Technical Report. 1996

[11] Gio Wiederhold, Peter Wegner, Stefano Ceri: Toward Megaprogramming. Communications of the ACM 35(11): 1992

[12] Gio Wiederhold. “Mediators in the architecture of future information systems.” IEEE Computer. Vol 25, No. 3. 1992.

[13] W. J. Tracz. Parameterized Programming in LILEANNA. Ph.D Thesis. Stanford University. August 1997

[14] M. Shan, R. Ahmed, J. Davis, W. Du, and W. Kent. Pegasus: A heterogeneous information management system. In W. Kim, editor, Modern Database Systems – The Object Model, Interoperability and Beyond, Addison-Wesley, 1995