Damia: Data Mashups for Intranet Applications

David E. Simmen, Mehmet Altinel, Volker Markl, Sriram Padmanabhan, Ashutosh Singh
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120, USA
{simmen, maltinel, marklv, srp, asingh}@us.ibm.com

ABSTRACT
Increasingly large numbers of situational applications are being created by enterprise business users as a by-product of solving day-to-day problems. In efforts to address the demand for such applications, corporate IT is moving toward Web 2.0 architectures. In particular, the corporate intranet is evolving into a platform of readily accessible data and services where communities of business users can assemble and deploy situational applications. Damia is a web style data integration platform being developed to address the data problem presented by such applications, which often access and combine data from a variety of sources. Damia allows business users to quickly and easily create data mashups that combine data from desktop, web, and traditional IT sources into feeds that can be consumed by AJAX, and other types of web applications. This paper describes the key features and design of Damia's data integration engine, which has been packaged with Mashup Hub, an enterprise feed server currently available for download on IBM alphaWorks. Mashup Hub exposes Damia's data integration capabilities in the form of a service that allows users to create hosted data mashups.

Categories and Subject Descriptors
H.m [**Information Systems**]: Miscellaneous

General Terms
Design

Keywords
Information Integration, XML, Data Feed

1. INTRODUCTION
There are two important trends motivating the need for a new type of enterprise information integration architecture, aimed primarily at satisfying the information integration requirements of situational business applications [1].

The first trend is happening inside the enterprise where there is an increasing demand by enterprise business leaders to be able to exploit information residing outside traditional IT silos in efforts to react to situational business needs. The predominant share of enterprise business data resides on desktops, departmental files systems, and corporate intranets in the form of spreadsheets, presentations, email, web services, HTML pages, etc. There is a wealth of valuable information to be gleaned from such data; consequently, there is an increasing demand for applications that can consume it, combine it with data in corporate databases, content management systems, and other IT managed repositories, and then to transform the combined data into timely information.

Consider, for example, a scenario where a prudent bank manager wants to be notified when a recent job applicant's credit score dips below 500, so that she might avoid a potentially costly hiring mistake by dropping an irresponsible applicant from consideration. Data on recent applicants resides on her desktop, in a personal spreadsheet. Access to credit scores is available via a corporate database. She persuades a contract programmer in the accounting department to build her a web application that combines the data from these two sources on demand, producing an ATOM feed that she can view for changes via her feed reader.

There are a large number of such situational applications being created by business users and departmental IT staff as an offshoot of solving day-to-day problems. These applications typically target a small community of users, and a specialized business need. In contrast, typical enterprise applications are developed by corporate IT staff for a large number of generic users, and a general purpose. Situational applications represent the "long-tail" of enterprise application development; consequently, there is a significant opportunity for IT researchers and professionals to create innovations that facilitate their development.

The second trend is happening outside the enterprise where the Web has evolved from primarily a publication platform to a participatory platform, spurred by Web 2.0 paradigms and technologies that are fueling an explosion in collaboration, communities, and the creation of user-generated content. The main drivers propelling this advancement of the Web as an extensible development platform is the plethora of valuable data and services being made available, along with the lightweight programming and deployment technologies those allow these "resources" to be mixed and published in innovative new ways.

Standard data interchange formats such as XML and JSON, as well as prevalent syndication formats such as RSS and ATOM, allow resources to be published in formats readily consumed by web applications, while lightweight access protocols, such as REST, simplify access to these resources. Furthermore, web-oriented programming technologies like AJAX, PHP, and Ruby on Rails enable quick and easy creation of "mashups", which is a term that has been popularized to refer to composite web applications that use resources from multiple sources [2].

The Damia project aims to seize upon the aforementioned opportunities to aid situational application development by harnessing many of the Web 2.0 paradigms and technologies that

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

have spurred the innovation in assembly manifested by the mashup phenomenon. We envision corporate IT steadily moving toward web style architectures. In particular, we envision the corporate intranet steadily evolving into a platform of readily consumable resources, and lightweight integration technologies, which can be exploited by business users to create "enterprise mashups" in response to situational business needs. The lines between the intranet and Web will progressively blur as enterprise mashups reach outside the corporate firewall to exploit data and services on the Web.

Enterprise mashups present a data problem, as they can access, filter, join, and aggregate data from multiple sources; however, these data machinations are often done in the application, mixed with business and presentation logic. In Damia, we are developing an enterprise-oriented data mashup platform on which such applications can be built quickly and easily, by enabling a clean separation between the data machination logic and the business logic. In particular, the Damia data mashup platform (1) enables secure access to data from a variety of desktop, departmental, and web sources both inside and outside the corporate firewall (2) provides the capability to filter, standardize, join, aggregate, and otherwise augment the structured and unstructured data retrieved from those sources (3) allows for the delivery of the transformed data to AJAX, or other types of web applications on demand (4) exposes these capabilities via lightweight programmatic and administrative APIs that allows users with minimal programming expertise to complete integration tasks.

We are still in the early stages of Damia's evolution; however, we have developed a rather sophisticated prototype of Damia's integration capabilities that we deployed in the context of a feed server, which was made available as a service on IBM's corporate intranet. In addition to exposing Damia's integration capabilities, which allowed users to create hosted data mashups, the service also provided a directory where the Damia community can tag, rate, and share data mashups, as well as other information assets that might be consumed by data mashups. A browser based user-interface exposed these capabilities in a way that allowed users with minimal programming expertise to take advantage of them.

The Damia data integration technology, along with key aspects of the original Damia feed server such as its user-interface and directory services design, have since been made available for download on IBM alphaWorks [3] in the context of Mashup Hub, an enterprise feed server that the Damia research team is jointly developing with IBM Software Group. Mashup Hub is a key technology of IBM's Information 2.0 initiative [4], which aims to extend the reach of the enterprise information fabric into the desktop, Web and other new data sources, and to provide tooling that facilitates situational application development.

1.1 Paper Organization

In this paper, we will describe the key features and design of Damia, focusing primarily on the data integration technology. The remainder of the paper is organized as follows. In section 2, we give an overview of the architecture of the Damia feed server that was deployed on the IBM Intranet[1]. In section 3, we present the data model, data manipulation operators, the data mashup compiler, and other details of the Damia integration engine. In section 4, we illustrate Damia's capabilities with use cases. Sections 5 and 6 discuss related and future work, respectively.

2. DAMIA FEED SERVER

This section provides a general overview of the Damia feed server architecture. The integration engine, which is the primary focus of the Damia project, is discussed in greater detail in Section 3.

The main components of the Damia feed server are depicted in Figure 1. In addition to the Damia integration engine, the server is further comprised of a directory services component, a storage services component, and a rich client browser interface.

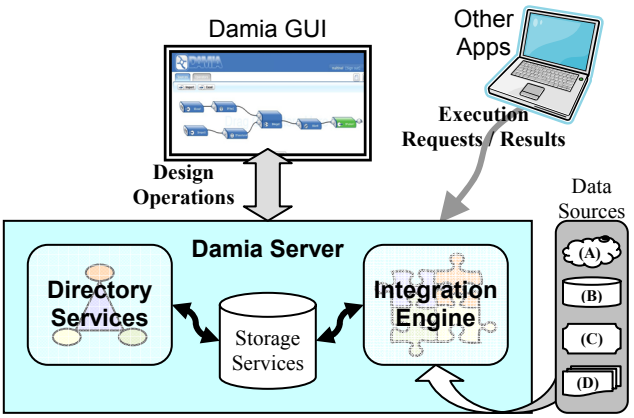

Figure 1 - Feed Server Architecture

2.1 Integration Engine

Damia provides a powerful collection of set-oriented feed manipulation operators for importing, filtering, merging, grouping, and otherwise manipulating feeds. These operators manipulate a general model of a feed, which can accommodate prevalent feed formats such as RSS and ATOM, as well as other XML formats. Data mashups are comprised of a network, or *flow*, of feed manipulation operators. The browser-based client interface provides a GUI for designing flows.

A flow is presented to the integration engine in the form of an XML document that depicts its flow representation in a serialized format. The flow is compiled into a set of lower level primitives that can be executed. A given feed manipulation operator might compile into number of lower level primitives.

A compiled data mashup has an associated URL; hence, the result of the data mashup can be retrieved via a simple REST call.

2.2 Directory Services

The feed server allows for general information assets to be cataloged, uploaded, and stored. Examples of information assets include data mashups, public spreadsheets, and URLs to interesting external feeds that might be used in data mashups. The directory services component provides capabilities to search, tag, rate, execute, and otherwise manage these information assets. In effect, it provides a Web 2.0 style framework for community-oriented information management. It also manages user profiles, authentication, and access control for assets. The directory services component uses the storage services component to store assets and metadata.

[1] The architecture of the original feed server we describe here closely resembles that of Mashup Hub

2.3 Storage Services

The storage services component handles the storage and retrieval of data and metadata needed by other Damia components. For example, it is used by the directory services component to store community information assets and associated metadata. The storage services component is also used by the metadata services component of the Damia integration engine to execute data mashups effectively.

2.4 Client Interface

Situational applications are typically created by departmental users with little programming knowledge; consequently providing an intuitive interface where data mashups can be composed visually was a critical design point for Damia. Toward this goal, we developed a browser-based user interface that provides facilities for composing, editing, and debugging data mashups graphically. Figure 2 shows a snapshot of the data mashup editor.

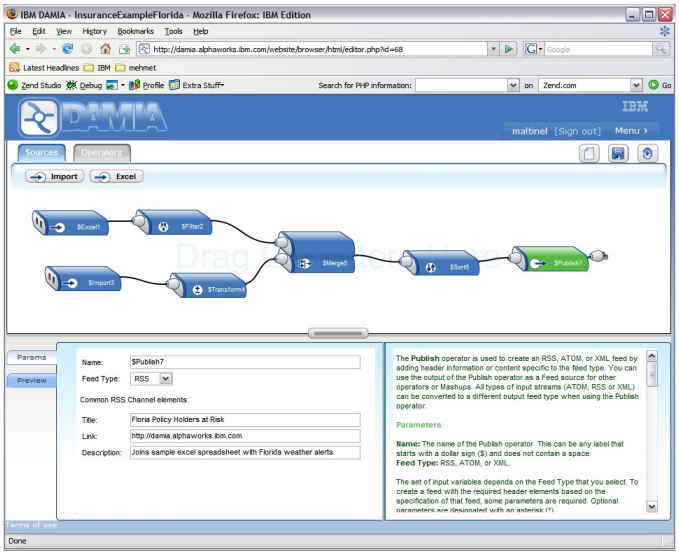

Figure 2 -Snapshot of the Damia GUI

The GUI allows users to drag and drop boxes, representing Damia operators, onto a canvas and to connect them with edges representing the flow of data between those operators. Users can use a preview feature to see the result of the data flow at any point in the process. This approach makes the development process more natural and less error prone. Once the data mashup design is completed, the result is serialized as an XML document and delivered to the server for further processing. It communicates with the server through a set of REST API interfaces, as illustrated in Figure 1. The client also provides an interface to directory services capabilities, thus allowing users to search for, and manage information assets from the same client interface that they use to compose data mashups.

2.5 Implementation

The Damia feed server was implemented with a LAMP stack. In particular, the integration engine, directory services, and other components are implemented in PHP. There were a few basic libraries from the LAMP ecosystem that we exploited. For example, we exploited the Zend Framework for caching. The storage services component makes use of a relational database to store resources and metadata. We currently support either DB2, or MySQL. The client interface is an AJAX application implemented using the Dojo toolkit [5]. The features of PHP used by the data integration engine are described in section 3.3.

3. DAMIA INTEGRATION ENGINE

The Damia integration engine compiles and executes data mashups. Figure 3 depicts the overall architecture and relevant APIs. The integration engine is comprised of a flow compiler, metadata services, and an augmentation engine.

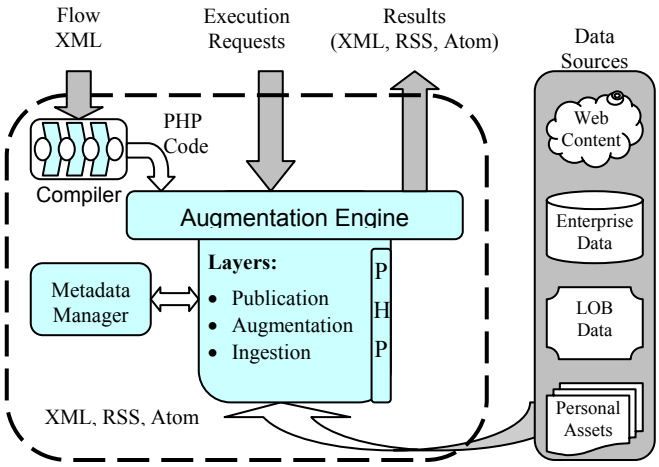

Figure 3: Damia Data Integrate Architecture

The *flow compiler* receives an XML specification of a data mashup and translates it into an augmentation flow, which is the manifestation of the data mashup that is executed by the augmentation engine. The XML specification represents the data mashup in terms of the conceptual feed-oriented data model and feed manipulation operators presented to end users; hence, it is the flow compiler that effectively implements the feed abstraction.

Metadata services stores the augmentation flow and the original XML representation of the data mashup, associating it with a resource identifier that can be used in subsequent execute, edit, and delete operations. Metadata services also manages metadata and functions needed by the flow compiler and augmentation engine in order to perform their tasks. For example, it manages ingestion functions, which are needed to map imported data resources to equivalent XML representations.

The *augmentation engine* executes a data mashup. It receives the resource identifier of the corresponding augmentation flow, which it retrieves from metadata services and evaluates. Typical result formats are ATOM, RSS, or general XML. The data manipulation primitives comprising an augmentation flow are implemented in PHP; thus, at its most basic level, an augmentation flow is simply a PHP script that is interpreted by a PHP engine.

The augmentation engine is conceptually divided into an ingestion layer, augmentation layer, and publication layer. The *ingestion layer* is responsible for importing data and for mapping imported data into an instance of the augmentation-level data model. It also contains a resource cache from which it can serve resources in order to avoid accessing data sources. The *augmentation layer* is responsible for manipulating instances of the augmentation-level data model in order to produce the data mashup result. It is comprised of augmentation operators that can evaluate xpath expressions; perform sorts, joins, grouping, construction, and low level manipulations. The *publication layer* is responsible for transforming

an instance of the augmentation-level data model to a specific format like RSS, ATOM, or JSON, which it then serializes for consumption by web applications.

The following subsections describe the data models, and the various components and layers of the Damia integration engine in greater detail.

3.1 Data Model

Feeds formats like RSS [6] and ATOM [7] are prevalent XML data interchange formats. Feeds often represent a set of data objects such as stock quotes, real estate listings, or employee records, which have been serialized for transport across a network. Damia provides a powerful collection of set-oriented *feed manipulation operators* for importing, filtering, merging, grouping, and otherwise manipulating feeds. The *feed-oriented data model* that forms the basis for such manipulation can easily represent standard feed formats like RSS and ATOM, but is designed to handle more general XML data formats that have repeating fragments that can be mapped to the model.

Data mashups are built by end users and applications in terms of the feed-oriented data model and feed manipulation operators; however, these are logical constructs that have no direct physical implementation. Feed manipulation operators are compiled into *augmentation operators*, which are lower level data manipulation primitives that can be executed by the integration engine. The *augmentation-level data model*, which forms the basis for data manipulation by augmentation operators, is a derivative of the Xquery data model (XDM) [8]. It is somewhat simpler than XDM, however. For example, it does not support node identity, or the full complement of data types.

3.1.1 Augmentation-level Data Model

The augmentation-level data model (ADM) is comprised of nodes, atomic values, items, sequences, and tuples. A *node* is the root node of a tree of nodes. A node can be an element node, attribute node, text node, or any other type of XDM node. An *atomic value* corresponds to an instance of a simple data type like a string, integer, or date. An *item* is either a node or an atomic value. A *sequence* is a named list of zero or more items. Finally, a *tuple* is a non-empty set of sequences.

Augmentation operators are closed under ADM. They consume one or more sets of tuples, or *tuple streams*, and produce a tuple stream. One class of operator works at the sequence level, extending tuples with new sequences derived from other sequences. For example, a *Union* operator creates a new sequence by combining the items of two or more sequences. Another class of operator works at the tuple level. For example, the *Sort* operator reorders the tuple stream based on the values of specified sequences that define the ordering key.

An *augmentation flow* is a network of augmentation operators that is executed using a data flow paradigm. The expressive power of an augmentation flow is analogous to that of an Xquery FLWOR expression. Consider Figure 4, which illustrates the relationship. It shows a simple Xquery FLWOR expression that creates a feed whose entries contain information about hotels in Vancouver joined with their reviews. Hotel information is coming from an ATOM feed provided by www.hotels.xyz, while hotel reviews come from an RSS feed provided by www.reviews.xyz.

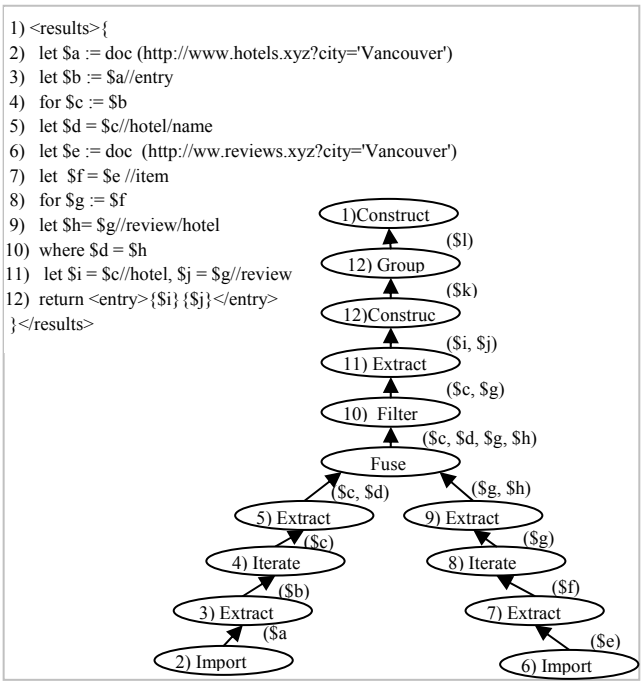

Figure 4 Augmentation Flow

Figure 4 depicts an abstract representation of an augmentation flow that is equivalent to the Xquery. The *Import*, *Iterate*, *Filter*, and *Construct* augmentation operators perform similar functions as the Xquery *doc* function, *for* clause, *where* clause, and *return* clause, respectively. The *Extract* operator evaluates xpath expressions. The graph shows sequence names flowing as tuples along the edges between augmentation operators. These sequences correspond to the sequence variables in the Xquery, which flow as binding tuples between FLWOR operations

The steps of retrieving the hotel feed, extracting feed entries, iterating each entry, and extracting the hotel name from an entry, as carried out in lines 2, 3, 4, and 5 of the Xquery, is implemented by the similarly numbered Import, Extract, Iterate, and Extract operators. Note that each of the operators projects sequences that are not needed in the data flow. For example, the Extract operator drops sequence $a, which contains the entire ATOM feed retrieved from www.hotels.xyz, after it has extracted all of the feed entries to form sequence $b.

The *Fuse* operator joins the tuple streams generated from the series of operations applied to the individual feeds retrieved from the two sources. The *Group* operator aggregates tuples into sequences, as is required in the example to turn the set of output tuples from the FLWOR expression into a sequence. In addition to implementing the Xquery return clause, the Construct operator is also used to construct the final result document from the result sequence. In general, a Construct operator is used to construct new sequences by substituting specified input sequences into a supplied XML template.

Damia has a number of other augmentation operators besides those required to implement a simple Xquery FLWOR expression. For example, the *Hsjoin* operator can perform a symmetric join comparable to a relational hash join.

Our objective is not to develop an implementation of Xquery in our augmentation layer, as we did not want to confine ourselves strictly

to the operations and semantics that the standard defines; however, there were definitely benefits to patterning our augmentation-level data model and corresponding augmentation operators after XDM and Xquery. Most importantly, it provided us with a strong semantic foundation for the manipulation of XML data. Further, it provided us with a guiding context in which to make extensions. For example, we came across a proposal for adding direct support for grouping to Xquery which helped influence the design of our *Group* operator [9]. Section 3.3.1 discusses the Group operator and other Damia augmentation operators in more detail.

3.1.2 Feed-oriented Data Model

The feed-oriented data model is comprised of containers, payloads, feeds, and tuples. A *container* is a special root node of a tree of nodes. Child nodes of a container node are restricted to element nodes; however, other nodes in the sub-tree rooted at the container node can be any XDM node. The term *payload* refers to the list of zero or more child nodes of a container node. A *feed* is a named list of zero or more containers. A *tuple* is a non-empty set of feeds.

The mapping of RSS and ATOM formatted XML documents to this model is straightforward, and handled automatically by Damia. One container node is created per RSS item, or ATOM entry, element. The payload of each container node is comprised of the element nodes that are children of the corresponding item or entry nodes. Damia feeds can be derived from other types of XML documents as well. All that is required is an xpath expression that identifies the payload. For example, the payload for a Damia feed that represents a set of real estate listings might be extracted from an XML document via the xpath expression *//listing*. Multiple instances of the feed-oriented data model can be extracted from the same document in this way. The process of mapping an XML document to a Damia feed maintains the payload in the same relative order as per the original document. The *Import-Feed* operator is responsible for mapping an XML document to an instance of the feed-oriented data model.

Feed manipulation operators are closed under the feed-oriented data model. Each operator consumes one or more feeds and produces a feed. For example, the *Filter-feed* operator takes a feed and a predicate as input, producing a new feed that is essentially a copy of the input feed, sans containers whose payload did not satisfy the specified predicate. A *Merge-feeds* operator, which is analogous to a symmetric relational join operation, concatenates payloads from two different input feeds that match according to a specified join predicate.

A *data mashup* is represented as a data flow graph of feed manipulation operators. Nodes in the graph represent operators, and edges represent the flow of feeds between operators. Feeds are packaged into tuples, which represent the unit of flow between operators. We introduced the notion of tuples into the feed-oriented data model in order to allow multiple versions of feeds to flow between operators. This capability enables different result feeds to be created for different subscribers, all in the same data mashup.

Feed manipulation operators are implemented by augmentation flows. Feeds manifest at the augmentation level as sequences whose items are rooted by a special container node. The child elements of the container node are its payload. In general, feed manipulation operators iterate over the container nodes of a sequence and perform filtering, joins, aggregation, and other set manipulations that involve the extraction and comparison of attribute and element values of the payload. The implementation of a data mashup is constructed by first expanding each individual feed manipulation operator into a subgraph of equivalent augmentation operators. A global optimization step is subsequently performed in order to transform the initial graph representation into one that can be executed more efficiently. Section 3.5 discusses feed manipulation operators in more detail. Section 3.6 details the process by which feed manipulation operators are compiled into augmentation flows.

3.2 Ingestion Layer

The Damia ingestion layer is comprised of connectors, ingestion functions, an Import operator, and a resource cache.

A *connector* is essentially a wrapper that implements an http interface to a particular data source. For example, a custom connector might be created to wrap "the DB2 database for department K55", or the "Acme Finance" website. Connectors understand the access protocols and APIs of the data source, they handle authentication, and take care of other detailed aspects of data source access. A connector can return files with any of the MIME types currently understood by the Damia ingestion layer. An example of a built-in connector is a simple file upload connector that allows users to upload spreadsheets, and other desktop data, thereby making it http accessible for reference in data mashups. The set of built-in connectors was designed to be extended by the Damia community.

An *ingestion function* maps a file of a specific MIME type to an XML representation. For example, Damia currently supports a built-in ingestion function for mapping comma separated files of MIME type text/csv to an ATOM feed representation. The set of built-in ingestion functions can also be extended by the Damia community.

The *Import operator* maps a resource provided by a data source into an instance of the augmentation-level data model by (1) invoking connectors or other web services to retrieve resources; (2) applying the appropriate ingestion function to a resource based on the MIME type of the resource; (3) parsing and mapping the XML representation of the resource into the data model. Although the Import operator is conceptually part of the ingestion layer, it supports the same iterator protocol as other augmentation operators, so we defer a more detailed discussion to section 3.3.

The *resource cache* provides basic caching of imported resources. It supports a simple *put* and *get* interface to insert and retrieve resources. The URL of the resource serves as the resource identifier for those calls. Caching is only in effect for the leaf nodes of a data mashup and the Import operator is the object that interacts with the cache. A cache policy requirement can be provided with a cache get call. We currently support only a freshness cache policy, which sets a limit, in terms of number of seconds, as to how stale a returned resource can be. More sophisticated policies will certainly be needed as we consider new scenarios. Resources are aged out of the cache in a cyclic fashion, with older resources evicted first.

3.3 Augmentation Layer

The Damia augmentation layer is comprised of a set of augmentation operators that are closed under the augmentation-level data model, as was described in section 3.1. Data mashups are compiled into augmentation flows, which are networks of augmentation operators that execute in a demand-driven data flow fashion. Augmentation operators produce and consume tuples according to an iterator protocol [10]. As such, each operator supports *bind-in*, *open*, *close*, and *next* interfaces, which are used by

other operators to initialize and uninitialize the operator, and to iteratively consume the tuple stream that the operator produces.

Operators have *arguments* which define input operands, as well as the operator's relationship to other operators in an augmentation flow. Augmentation operators execute within a *bind-in context* which provides values for variables that are referenced in arguments. For example, a URL parameter that returns the price for a given stock would contain a variable reference to the stock in the URL, which it would pull from the bind-in context. The bind-in context for all operators is initialized with the attribute-value pairs in the HTTP context passed to the data mashup when it is executed; hence, variables can be passed to operators from outside the data mashup. Nested loop operations can extend the bind-in context of inner loop operators with attribute-value pairs passed from the outer loop. An operator's bind-in method, which provides and operator with its bind-in context, must be executed before an operator is opened.

3.3.1 Implementation

Augmentation operators are implemented as PHP classes. Data mashups are essentially compiled into PHP scripts that are interpreted by the PHP engine. The augmentation operators rely on a couple of basic PHP features such as PHP DOM support for parsing XML documents and for executing xpath statements, as well as PHP Curl libraries for importing resources via HTTP.

Figure 5 shows PHP pseudo code of an augmentation flow, which joins stocks from a specified portfolio with the current stock prices. Operators are wired together, and get their arguments; by setting class properties. Every operator class provides methods to set arguments. For example, the *setInOp* method of Iterate identifies the Extract operator as its input operator. All classes implement the bind-in, open, next, and close methods of the iterator protocol. To start the execution flow, the bind-in, open, and next methods of the topmost operator of the augmentation flow (the Construct operator in the example) are executed. Such requests cascade down through the augmentation flow, tuples flow from the Import operators back up through the augmentation flow, and the execution continues until all tuples are drained from the pipeline. Variable references use PHP naming conventions (e.g. $pname variable in line 2). The values for these variables are pulled from the bind-in context. In this example, the bind-in context is initialized from the HTTP context (see line 22).

```
1) $import1 = new Import ();
2) $import1()->setURL(
'http://myportfolio.com?name=$pname')
;
3) $import1()->setOutSeq('$assets');

4) $extract = new Extract();
5) $extract->setInOp($import1);
6) $extract->setXpathEntry(array(
'$assets'  ,'//iterate/text()', $tickers'));

8) $iterate = new Iterate();
9) $iterate->setInOp($extract);
10) $iterate->setInSeq($tickers);
11) $iterate->seOutSeq($ticker);

12) $import2 = new Import ();
13) $import2->setURL(
'http://stockprice.com?ticker=$ticker');
14) $import()->setOutSeq('$prices);

15) $fuse = new Fuse();
16) $fuse->setOuterOp($iterate);
17) $fuse->setInnerOp($import2);

18) $cons = new Construct();
19) $cons->setInOp($fuse);
20) $cons->setTemplate(
'<res><tick>$ticker</tick>$prices</res>')
;
21) $cons->setOutSeq($construct);

22) $cons->bindIn($_REQUEST);
23) $cons->open();
24) while ($tuple=$cons->next()
                                != EOS) {
25) echo ($tuple->getSequence($cons)->
getItem(0)->serialize()) . '\n'); }
```

Figure 5 -PHP Representation of Augmentation Flow

3.3.2 Augmentation Operators

This section offers a short description of the augmentation operators currently supported by Damia:

Import

The Import operator maps a resource from a data source into an instance of the augmentation-level data model (ADM). It is analogous to an Xquery doc function. The Import operator takes a *resource*, *cache policy,* and *output sequence* as arguments. The resource argument represents the resource to be imported. It can be a string, file path, or a URL. The Import operator imports the specified resource and makes a call to the metadata services component to retrieve the appropriate ingestion function for rendering the resource into an XML representation. The step of retrieving a URL specified resource is handled by the PHP Curl library. This step can be avoided if the cache manager has a cached version of the resource that satisfies the specified cache policy. The imported and transformed resource is mapped to a PHP DOM and associated with the sequence name provided by the output sequence argument.

Iterate

The Iterate operator iterates over each item of a target sequence. It is analogous to the for clause of an Xquery FLWOR expression. The Iterate operator takes an *input operator*, *input sequence*, and *output sequence* as arguments. The Iterate operator iterates each item of the input sequence, producing one new tuple per item. Each output tuple contains all sequences of the input tuple, plus a new sequence representing the iterated item. The output sequence argument provides the name of this new sequence.

Filter

The Filter operator drops tuples from a tuple stream that do not satisfy a specified predicate. It is analogous to the where clause of an Xquery FLWOR expression. The Filter operator takes an *input-operator* and *predicate* as arguments. The specified predicate compares sequences of the input tuple using Xquery semantics. For example, the predicate $S = 10, where $S is the names of a sequence of the input data stream, would qualify tuples based on existential semantics.

Extract

The Extract operator applies a set of xpath expressions to a tuple stream. The Extract operator takes an *input operator* and *xpath array* as input. The xpath array argument is an array that supplies an *input sequence*, *xpath expression*, and *output sequence* with each entry. The Extract operator applies all xpath expressions in the xpath array argument. The xpath expression attribute of each array entry provides the xpath expression to apply, while the input sequence attribute indicates the sequence to apply it to, and the output sequence attribute indicates the name of the result sequence.

Expression

The Expression operator generates new sequences by evaluating a set of expressions. It takes an *input operator*, and an *expressions array* as input. Each entry in the expressions array identifies an *expression* to apply, and the *output sequence* names the result. For example, an expression of the form COUNT($S), where $S is a sequence of the input tuple, would return an output sequence that represented the number of items in $S. Currently, the classes of expressions supported include (1) aggregate functions like sum, and count (2) string functions that perform concatenation, form substrings, or evaluate regular expressions.

Construct
The Construct operator generates a new sequence according to an XML template. It is analogous to the return clause of an Xquery FLWOR expression. The Construct operator takes an *input operator*, *template*, *substitutions array*, and *output sequence* as input. Each input tuple is extended with a new sequence formed by substituting nodes and values from input sequences, into the output template. Information in the substitutions array argument identifies how to make the substitutions. It specifies input sequence names and associated xpath expressions that identify locations in the template where the substitution should be made.

Group
The Group operator partitions and aggregates a tuple stream according to specified grouping key. The Group operator takes an *input operator*, *group sequences array*, and *nest sequences array* as input. It partitions the tuples of the input data stream according to the values of input sequences identified by the group sequences argument, and aggregates tuples of a partition by combining the sequences identified in the nest sequences array argument. The sequences identified in group sequences array must have a single item. The string values of those items together determine the partition that the tuple belongs to. The Group operator returns a single output tuple per partition. If the group sequence array is empty, the group operator collapses the data stream into a single tuple. If the nest sequences array is empty, it does the equivalent of a distinct operation with respect to the grouping key sequences.

Sort
The Sort operator sorts a tuple stream according to specified sorting key. The Sort operator takes an *input operator*, and *sort sequences array* as input. The sort-sequences array identifies the input sequences whose values will form the sort key, plus an ascending or descending ordering attribute. The Sort operator exploits built in PHP sort routines. The sort key values are extracted from an input tuple similar to how the Group operator extracts partitioning key values.

Fuse
The Fuse operator joins two input tuple streams using a basic nested-loops join algorithm. The Fuse operator takes an *outer loop operator* and *inner loop operator* as input. The operator identified by the inner loop operator argument is executed in the context of each tuple of the outer loop tuple stream. The inner loop context is formed by adding the sequences of the current outer loop tuple to Fuse's bind-in context. Tuples produced in the context of a current outer tuple are concatenated with the outer tuple. Multiple tuples might be returned per inner tuple. The Fuse operator is used to drive access to sources that depend on another source for input values. For example, a review site providing hotel reviews might require a hotel name as a URL parameter. A Fuse operator would progressively drive an Import operator that returned reviews for a given hotel.

Hsjoin
The Hsjoin operator joins two tuple streams using a simple hash join algorithm. The Hsjoin operator takes a *build operator*, *probe operator*, *build sequences array*, and *probe sequences array* as input. The build operator argument identifies the operator providing the tuples to build the hash table. The probe operator argument identifies the operator providing the data stream that probes the hash table. The sequences used to form the hash keys for the build data stream and probe data stream are provided by the build sequences and probe sequences arguments, respectively. The hash keys are formed from input sequences in the same way that the grouping key is formed by the Group operator.

Union
The Union operator forms a new sequence by appending the items of multiple input sequences. The Union operator takes an *input operator*, and *union sequences array,* and an *output sequence name* as input. The union sequences array identifies the input sequences whose items are appended together to form the output sequence.

3.4 Publication Layer
The publication layer transforms an instance of the augmentation-level data model into a specific representation, like ATOM, RSS, CSV, or JSON, and then serializes the result for HTTP transfer. Currently, the publication layer uses construction as the means for mapping from our internal sequences representation to the specific target representation. For this, it relies on a set of standard templates made available through metadata services (e.g. there is standard construction template for formatting ATOM feeds). The publication layer has recently undergone a major enhancement related to newly added support for streams, continuous data mashups, and syndication, which is currently prototyped but not yet available for download.

3.5 Feed Manipulation Operators
Feed manipulation operators are closed under the feed-oriented data model described in section 3.1.2. Data mashups are comprised of a network of feed manipulation operators that execute in a data flow fashion. The operators have *arguments* which define input operands, and their relationship to other operators in the data mashup. Feed manipulation operators operate on one or more feeds and produce a feed. The names of the input and output feeds are provided as arguments. If the name of the output feed is different from that of the input feeds, then the operator essentially creates a new version of the input feed. All versions of feeds flow in a single tuple from one operator to the next. This section provides a brief description of the feed manipulation operators which are currently available. The compilation process whereby these logical operators are translated into augmentation flows is described in section 3.6.

Import-Feed
The Import-feed operator imports XML data into an instance of the feed-oriented data model. It takes a *feed type, source URL*, *repeating element*, and *cache policy* as input. The source URL identifies the XML resource to import. The feed type argument can identify XML, RSS, or ATOM. If XML is specified, then the repeating element argument is used to identify the payload from the input XML document. The Import-feed operator has built-in capability to extract payload for RSS and ATOM formatted input. The cache-policy parameter indicates the freshness criteria for serving data from the ingestion layer cache.

Filter-Feed
The Filter-feed operator removes containers from a feed that fail to satisfy a filter condition. It takes an *input operator, input feed name, filter condition array*, and *output feed name* as arguments. The filter condition array argument identifies a set of *filter conditions* that are applied as conjuncts. Each filter condition specifies an *xpath expression*, *comparison operator*, and a *value*. The xpath expression identifies the data in each container that is compared to the value using the comparison operator.

Merge-Feeds

A Merge-feeds operator is analogous to a symmetric relational join operation. It concatenates payloads from two different input feeds that match according to a specified merge condition. It takes a *left operator*, *left feed name, right operator, right feed name*, *merge condition array*, and *output feed name* as arguments. The left feed name and right feed name identify the input feeds being merged. All combinations of left feed and right feed containers are compared. The merge condition array identifies a set of *match conditions* that are applied as conjuncts. Each match condition specifies a *left xpath expression*, *comparison operator* and *right xpath expression*. The left xpath expression and right xpath expression identifies the data from the left feed and right feed payloads that are compared using the comparison operator. Payloads from container combinations that match are concatenated into an output feed container.

Augment-Feed

An Augment-feed operator concatenates payloads of an outer feed container with inner feed containers produced in its context. It takes an *outer operator*, *outer loop feed name, inner operator, inner loop feed name*, *bindings array*, and *output feed name* as arguments. The Augment-feed operator works similar to a Fuse augmentation operator, but with feed entries, as opposed to tuples, as the unit of manipulation. The bindings array is used to form the bind-in context for the inner operator. It consists of a set of xpath expressions that are applied to the current outer feed container. One result container is constructed per combination of outer feed container and inner feed container produced in its context.

Transform-Feed

The Transform-feed operator restructures an input feed using a specified template. It takes an *input operator, input feed name, template*, *substitutions array, bindings array, expressions array*, and an *output feed name* as arguments. It is analogous to mapping the return clause of an Xquery FLWOR expression over each input feed container. The template and substitutions array play a similar role here as they do in the Construct augmentation operator discussed in section 3.3. The bindings array is used to extract values from the payload of each input feed container. The expressions array computes additional values using these extracted values. The set of values computed from extraction and expression computation are then substituted into the template according to the substitutions array.

Group-Feed

The Group-feed operator partitions the payload of incoming feed containers according to specified grouping key bindings, producing one feed entry per partition, which is formed by concatenating the corresponding payload of containers in the same partition. It takes an *input operator, input feed name, group key bindings*, *nest key bindings,* and an *output feed name* as arguments. Group-feed is analogous to the Group augmentation operator but with feed containers the unit of manipulation. The group key bindings are extracted from each input feed container to extract the values that determine the output partition for the container. The nest expression bindings determine which fragments of the payload are extracted and added to the corresponding output feed entry for the partition.

Sort-Feed

The Sort-feed operator reorders the containers of an incoming feed according to a specified sort key. It takes an *input operator, input feed name, sort key*, and an *output feed name* as arguments. Sort-feed is analogous to the Sort augmentation operator but with feed containers the unit of manipulation. The sort key is comprised of a list of *sort expressions* and *ordering attributes*. The sort expression is basically an xpath expression that extracts a value for a sort key component from a payload. The ordering attribute determines if containers are ordered in ascending or descending order relative to that sort key component.

Union-Feed

The Union-feed operator forms a single output feed by appending the containers of multiple input feeds. It takes an *input operators array, input feeds array,* and *output feed name* as arguments. The Union-feed operator first concatenates all input tuples coming from the operators specified in the input operators array. It then takes the feeds specified by the input feed array and appends their containers to form the output feed.

Publish-Feed

The Publish-feed operator transforms an instance of the feed-oriented data model into a specific XML format. It takes an *input operator name, input feed name, output feed type, feed bindings*, and *output feed name* as arguments. The feed type can specify a format such as RSS, ATOM, or XML. The feed bindings supply feed header data for the output feed, which depends upon the specified feed type. For example, an ATOM feed has "title" and "id" fields. The operator emits all container items in a feed format, with header and repeating containers. There are standard templates for RSS and ATOM. For XML feed types, users provide the header and specify the element name used for repeating container elements.

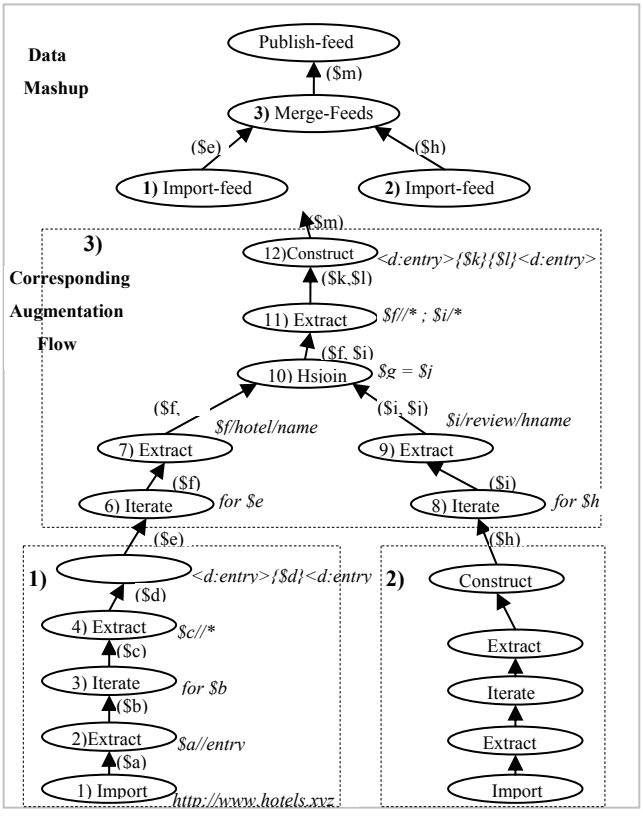

Figure 6 Data Mashup and Augmentation Flow

3.6 Flow Compiler

The Damia compiler is responsible for mapping a data mashup comprised of feed manipulation operators into an augmentation flow. Figure 6 illustrates the relationship between feed manipulation operators and augmentation flows. It depicts a data mashup representation that implements the Xquery expression in Figure 4, as well as augmentation flow representations that correspond to the feed manipulation operators in the data mashup. The Import-feed operators of the data mashup are responsible for mapping the ATOM and RSS feeds containing hotel and review information into an instance of the feed-oriented data model. The Merge-feeds operation concatenates the payloads of the imported feeds that agree on hotel name. The Publish-feed operator transforms the output of the Merge-feeds operator from its representation in the feed-oriented data model, to a serialized format that can be readily consumed by web applications, such as RSS or ATOM.

The augmentation flow subgraph that implements a given feed manipulation operator is contained in the rectangle that shares the same number as the corresponding operator. Edges are annotated with the names of the sequences produced by an operator. Annotations next to an operator indicate arguments to the operator. Import-feed operator (1) is implemented by the Import (1), Extract (2), Iterate (3), Extract (4), and Construct (5) augmentation operators. This sequence of operations maps the ATOM feed imported from www.hotels.xyz into sequence $e, which represents an instance of the feed-oriented data model. The mapping is accomplished by iterating each entry of the imported ATOM feed, and essentially moving the entire payload of each entry under a special container node, denoted as <d:entry> in the example. A similar series of augmentation operators implements Import-feed operator (2), which maps the RSS feed from www.reviews.xyz into sequence $h, which represents another instance of the feed-oriented data model.

The Merge-feeds (3) operator is implemented by iterating over the containers of sequences $e and $f, which were produced by the operators corresponding to Import-Feed (1) and Import-feed (2), extracting hotel names from their payloads into sequences $g and $j, and using the Hsjoin operator to join tuples that contain input feed entries which agree on hotel name. The Extract (11) and Construct (12) operators then map the payloads of the matching input feed entries from sequences $f and $i, into a new result container.

3.6.1 Flow Compiler Overview

The flow compiler takes an XML document describing the data mashup, and emits a PHP script comprised of augmentation operator classes organized into an augmentation flow. The flow compiler consists of three modules as shown in Figure 7. A *parser* module parses the XML representation of a data mashup into an *augmentation graph* representing an initial augmentation flow. A *rewrite* module transforms the augmentation graph into a more efficient representation. Finally, a *code generation* module traverses the augmentation graph and emits a PHP script.

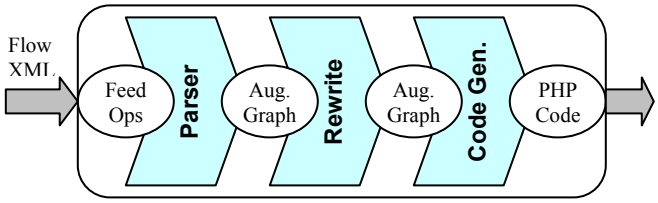

Figure 7: Mashup and Augmentation Flow

3.6.2 Parser Module

A data mashup is presented to the flow compiler as a set of XML elements that represent feed manipulation operators. Connections between operators are represented using id references, i.e. the structure of the document is flat. Other attributes and child elements are understood by the parser as arguments of the feed manipulation operator. The parser translates each element into a subgraph of the initial augmentation graph. Each node in the graph represents a single augmentation operator. Figure 6 illustrated the correspondence between feed manipulation operators and a subgraph of augmentation operators.

3.6.3 Rewrite Module

The rewrite module performs a series of rule-based transformations to the initial augmentation graph. The rules are currently based on heuristics. There are currently no cost-based transformations. The rule engine is extensible. New rules are added by implementing an abstract class that defines common methods and properties. Rules can be turned on or off. The current prototype implements a small number of rewrite rules. For example, there is a rewrite rule for removing redundant sequences of Group and Iterate operators, which result from the myopic expansion of an individual feed operator into an augmentation subgraph. Figure 8 illustrates the scenario.

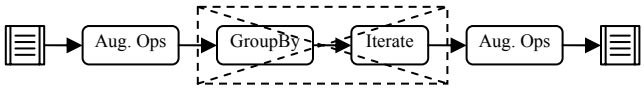

Figure 8: Rewrite Rule Example

We are planning to extend this core set of rewrite rules as we observe more common usage patterns.

3.6.4 Code Generation Module

The code generation module performs a topological sort of the augmentation graph to form an ordered list of augmentation nodes. It then generates an augmentation operator class instance per list entry, setting properties of each class instance in order to wire the operators together, and to set other input arguments, as per the example shown in Figure 5. All code segments are then appended, and appropriate headers added, to form the final script.

3.7 Integration Engine Interfaces

All interfaces to the integration engine are made available via a REST protocol. Damia supports client interfaces for compiling a data mashup, previewing the results of a partially built data mashup, and for executing a compiled data mashup. Damia also supports administrative interfaces that provide metrics on data mashup execution, server diagnostics, and so on. Such metrics are maintained as feeds that can be used as sources for data mashups.

4. USAGE SCENARIOS

In this section, we examine two usage scenarios that illustrate how Damia enables business and departmental users to integrate and visualize enterprise, departmental, and external data sources.

4.1.1 Customer Service Application

Customer service is a familiar enterprise application. The customer service application is typically a pre-built package with well defined interfaces for accessing appropriate data and for logging the details of each customer interaction. For our scenario, imagine that Cassie is a customer service representative for JK Enterprises who is fielding a call from customer Bob regarding a dispute in his credit billing statement. Apparently, Bob has been billed twice on a purchase of an appliance item and is calling to complain about this error. In this scenario, Cassie needs to perform the following tasks:

- Validate the customer details
- Pull up the customer profile and transaction details
- View the billing statements
- Validate the dispute
- Process a dispute resolution workflow

Now, imagine we are adding new features to the application in order to verify customer information faster or to provide specific promotions to the customer. For example, IBM has introduced a new capability called IBM Global Name Recognizer (GNR) which can find matching names from a dictionary of names. How might this capability be introduced into the application?

Damia technology provides an easy way to enable this extension. The first step is to create a REST service from GNR that generates a list of phonetically matching names given a roughly spelled input name. For instance, if Cassie was to type "Denis Mastersen", the GNR service might return the entries in Figure 9.

```
<Names>
    <Name> Dennis   Masters </Name>
    <Name> Dennis    Masterson </Name>
    <Name> Denise    Masterson</Name>
    <Name> Denise   Masters </Name>
    <Name> Deny  Masterton </Name>
</Names>
```

Figure 9 GNR Service Output Feed

Once we have this service available, the next step is to create a Damia data mashup that can combine GNR service output with the matching customer accounts in JK Enterprise's customer database. Figure 10 shows the mashup that performs this task.

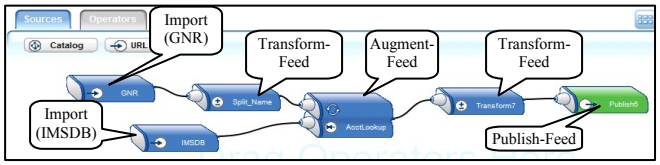

Figure 10: Damia Mashup of GNR and Customer Data

Cassie is expected to enter the rough customer name as input to the GNR source operator in the data mashup. The GNR service returns the list of matching names similar to what is shown in Figure 9. The Damia data mashup then massages this output, extracts the relevant last and first names and performs a lookup against the JK Enterprises customer database. The names that match in the customer database are published in an output feed, shown in Figure 11. Cassie can then choose the matching customer using attributes such as address or phone number and proceed to her next task.

```
- <rss version="2.0">
  - <channel>
      <title>Customer Lookup</title>
      <link>http://www.jkstore.com/</link>
      <description>Account for customers with similar names</description>
    - <item>
        <out_first_name>John</out_first_name>
        <out_last_name>Smith</out_last_name>
        <out_street>Almaden Lake Lane</out_street>
        <out_account_number>8885221</out_account_number>
      </item>
    - <item>
        <out_first_name>John</out_first_name>
        <out_last_name>Smyth</out_last_name>
        <out_street>Homestead Road</out_street>
        <out_account_number>5691903</out_account_number>
      </item>
    </channel>
  </rss>
```

Figure 11 Customer Feed with GNR Lookup Capability

4.1.2 Real Time Situational Monitoring

Real-time situational applications (emergency co-ordination or tracking) are very good candidates for Damia mashup capabilities. In the following example, we consider the situation where a hurricane is bearing down on a region and an insurance agent is assessing the potential claim damage exposure from this hurricane. We believe that there are many situational applications similar to this one (imagine the home improvement retailers in the same region etc.) who would be able to work with such a technology to improve their forecasting or response capabilities.

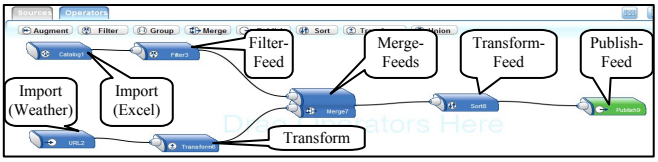

Figure 12: Weather Alerts and Insurance policy data

In this insurance scenario, the agent launches her mashup application upon notice of any severe weather alerts. Figure 12 shows the mashup flow. The agent takes her client policy information from a personal spreadsheet database. We note that personal information that might be in an analyst's or agent's workspace is important information that needs to be integrated into mashup applications. In Damia, this is done by uploading such spreadsheets into the data mashup server and using these spreadsheets as data feed sources. The second source input for this mashup is the National Weather Service which provides a feed of severe weather alerts for a particular region or state. In our example, the data mashup extracts city codes from the spreadsheet and the weather feeds and performs a merge operation. The output is a list of insurance policy holders who are likely to be affected by the severe weather. This output can then be used to show a rich interactive view of affected cities and estimated personal property damage. Figure 13 shows a rendering of such a rich interactive application using JustSystems XFY [11] tool that utilizes the Damia feed output. In this figure, notice the storm tracker can be manipulated by the agent so that the storm path changes can trigger the processing of the Damia mashup in order to present a concise output of the potential storm damage. This storm damage is shown visually using the map and charting widgets.

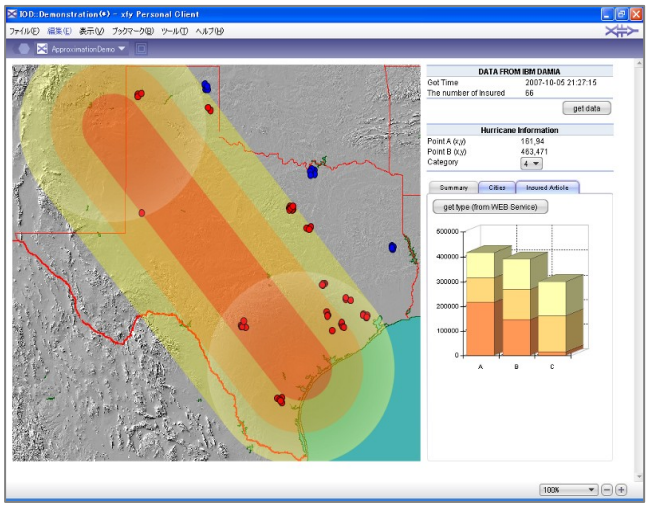

Figure 13: Storm Tracker Application with Damia Input

5. RELATED WORK

Traditional enterprise integration architectures, such as EII and ETL architectures, are not designed to satisfy the integration requirements of enterprise mashups. First of all, these architectures were designed primarily to handle a large amount of structured data made available by a small set of familiar and fairly static enterprise sources. Secondly, they typically require the services of a highly knowledgeable information architect capable of defining complex schema mappings, views, or ETL flows. Conversely, an enterprise mashup platform is comprised of a large and dynamically changing set of data sources. The dynamic nature of a web platform makes it especially hard to impart the necessary structure and semantics to data before it is manipulated. Moreover, the developers of enterprise mashups are not information architects, but departmental users with minimal programming expertise. Damia is designed for a large and changing number of data sources that make available both structured and unstructured information, and to enable less skilled users to complete complex integration tasks.

Damia is focused on the data problem presented by situational applications; hence, it uses a data flow model to represent a mashup, leaving the control flow aspects to the application. In contrast, the Google Mashup Editor [12] and Microsoft PopFly [13] bring application logic into the equation along with the data manipulation logic. The Mash-O-matic [14] project focuses more on the behavioral aspect of assembling mashups by enabling mashup builders to select input data from web pages. Intel's MashMaker [15] has a principled programming model based on a functional language that allows users to create mashups while surfing the web. None of these projects focuses squarely on the data aspects of the problem.

To our knowledge, the only other service that provides a data flow oriented platform is Yahoo Pipes [16]. Pipes allows data mashups that combine RSS, ATOM, or RDF formatted feeds. Further, it provides a GUI that allows data mashups to be specified graphically. Pipes focuses on the transformation of feeds via web service calls (e.g., language translation, location extraction). Damia goes beyond Yahoo Pipes in several ways: (1) Damia has a principled lower level data model for manipulating XML data and a more general feed manipulation model for composing data mashups (2) Damia is targeting enterprise business users. As such, it must support a more powerful set of data manipulation operators such as those that do general joins, aggregation, and other sophisticated transformations that Yahoo Pipes does not offer. (3) Damia has to deal with enterprise sources and data types such as Lotus Notes data (email), Excel spreadsheets, and corporate relational databases, in addition to the web feeds the Yahoo! Pipes focuses on. (4) Security is a major concern of Damia. Access control and authentication are required for premium sources inside the firewall, as well as those emerging on the web through information marketplaces like StrikeIron [17].

6. FUTURE WORK

The Damia project is progressing simultaneously on both research and product fronts. The research team has developed a blueprint for web style data integration that is guiding its current agenda. This section describes some of the current focus areas.

Data Standardization

Data standardization is an important current focus of ours, as the quality of any integration engine is reflected by how effectively it can combine data from different sources. Toward this goal, we are seeking to exploit text annotation services (e.g. ClearForest [18], UIMA [19]), master data management technology (e.g. IBM Master Data Management Server [20]), and other available services (e.g. Yahoo! Geocoding API [21]) both inside and outside the corporate firewall in order to add structure to feeds, and to reconcile differences in how entities are represented by the diverse data sources that Damia aims to support. One of the distinguishing aspects of our approach lies in our use of folksonomy to attain the metadata used in performing entity resolution. This technique allows us to harness the collective intelligence of the Damia community in order to progressively improve the fidelity of our data standardization algorithms.

Continuous Data Mashups

Another area of early focus has been on "publish and subscribe" enterprise mashup scenarios wherein data mashups run continuously, syndicating new results to subscribers whenever relevant data is available from input sources. Toward these goals, we have recently added support for streams, windows, continuous data mashups, and subscribers to Damia.

Data Ingestion

Data fodder for enterprise data mashups exists in forms other than nice web feeds, such as office documents, email, relational databases, and HTML pages. It is critical to provide tooling that facilitates access to such data. We have recently prototyped a general purpose "screen scraping" (e.g. Kapow [22], Lixto [23]), engine that produces feeds from the valuable data locked within HTML pages. An important next step involves making the scraper self-repairing in response to changes in web page format.

Search

Mashup applications are created by large numbers of small communities. An effective search capability must be provided in order to promote sharing of data feeds amongst communities. We believe that such a search mechanism must not only look at metadata associated with a data mashup, but it must also look at its data and data manipulation logic.

Data Quality

What kind of business decisions should one make based on a feed that combines a premium Dun and Bradstreet financial feed with John Doe's blog of financial musings? We believe that it is

important to make lineage information available with a feed so that users might assess its quality.

7. CONCLUSION

There is a significant market opportunity for technology that can help business leaders exploit information from desktops, the web, and other non-traditional enterprise sources, in order to react to situational business needs. IBM's Information 2.0 initiative [4] is targeting that opportunity. It aims to provide technology that can help extend the reach of the enterprise information fabric and help IT departments provision and manage situational applications built by enterprise business users. The initiative includes Mashup Hub, an enterprise feed server that facilitates the creation and management of data feeds that can be used by situational applications. The Damia data integration engine described in this paper is a key component of Mashup Hub [3]. Damia enables the creation of data mashups that combine data from desktop, web, and traditional IT sources into feeds that can be consumed by AJAX, and other types of web applications. Damia presents data mashup creators with a set of data manipulation operators that allow data mashups to be created from a data flow perspective, around a generalization of the familiar notion of a feed. A set of connectors, ingestion functions, and powerful data manipulation operators based on a principled XML data model; provide the underlying implementation of this data mashup abstraction. The Damia data integration technology is currently available for download on IBM alphaWorks via Mashup Hub.

8. ACKNOWLEDGMENTS

We thank Paul Brown, Susan Cline, Ken Coar, Sunitha Kambhampati, Rajesh Kartha, Eric Louie, Sridhar Mangalore, Louis Mau, Yip-Hing Ng, Kathy Saunders, and the other IBM colleagues that helped develop Damia. Special thanks to Anant Jhingran and Hamid Pirahesh for valuable discussions. Thanks also to Fatma Ozcan for reviewing an early draft of the paper.

9. REFERENCES

[1] A. Jhingran, “Enterprise Information Mashups: Integrating Information, Simply”, VLDB 2006: 3-4.

[2] Programmable Web, http://www.programmableweb.com/

[3] IBM Mashup Starter Kit http://www.alphaworks.ibm.com/tech/ibmmsk

[4] IBM Info 2.0 http://www-306.ibm.com/software/data/info20/

[5] Dojo, the Javascript toolkit, http://dojotoolkit.org/

[6] RSS http://cyber.law.harvard.edu/rss/rss.html

[7] The ATOM Syndication Format, http://tools.ietf.org/html/rfc4287

[8] M. F. Fernandez, A. Malhotra, J. Marsh, M. Nagy and N. Walsh, “XQuery 1.0 and XPath 2.0 Data Model ”, January 2007, http://www.w3.org/TR/xpath-datamodel/

[9] K. S. Beyer, Kevin S. Beyer, D. D. Chamberlin, L. S. Colby F. Ozcan, H. Pirahesh, Y. Xu, "Extending Xquery for Analytics" SIGMOD Conference 2005: 503-514

[10] G. Graefe "Query Evaluation Techniques for Large Databases" ACM Comput. Surv. 25(2): 73-170 (1993)

[11] Just Systems http://www.xfy.com

[12] Google Mashup Editor, http://code.google.com/gme/

[13] Microsoft PopFly, http://www.popfly.ms/

[14] Mash-o-Matic, http://sparce.cs.pdx.edu/mash-o-matic/

[15] Intel Mash Maker, http://mashmaker.intel.com/

[16] Yahoo Pipes, http://pipes.yahoo.com/pipes/

[17] Strikeiron Inc., http://www.strikeiron.com/

[18] Clearforest Inc., http://www.clearforest.com/

[19] Unstructured Information Management Architecture (UIMA), IBM Research, www.research.ibm.com/UIMA/.

[20] IBM Master Data Management Server www306.ibm.com/software/data/ips/products/masterdata/

[21] Yahoo! Maps Geocoding API http://developer.yahoo.com/maps/rest/V1/geocode.html

[22] Kapow Technologies, http://www.kapowtech.com

[23] Lixto Software, http://www.lixto.com/