A Layered Architecture for the Exploration of Heterogeneous Information Using Coordinated Views

Isabel F. Cruz Yuan Feng Huang
Department of Computer Science
University of Illinois at Chicago
{ifc,yhuang}@cs.uic.edu

Abstract

In a real-world decision support application, users often want to search data from various sources according to some criteria, build a visualization based on the data being retrieved, and use the visualization to explore the data. With our approach, these activities are supported within the same workspace. Views are dynamically created by binding each data set to a visualization template according to the user's preferences. The resulting views are then arranged into a larger coordinated view. In our layered architecture, data flows through the layers becoming encapsulated inside of metadata that describes the visual attributes being added. This metadata determines both the individual views and the dynamic interactions within a coordinated view. Dynamic interactions are implemented using a mediated notification services architecture.

1. Introduction

There are several important issues related to the exploration of information such as: integrating and retrieving data from heterogeneous data sources, building visualizations of the information, coordinating different visualizations that reflect different aspects of data, and dynamically assembling visualizations based on the user's preferences. In this paper we describe an approach that addresses these issues. In our approach, information integration activities and the visual exploration of the information occur in a single workspace, supporting:

- A robust and transparent way to access the information from heterogeneous databases.
- A generic, interactive, and dynamic visualization framework supporting customization, multiple integrated views, and the coordinated exploration of the data.
- A single framework where information access and visualization are fully integrated.

We have designed a framework and fully implemented a prototype based on an architecture that combines the information visualization and information integration aspects of the exploration of heterogeneous information. Our framework extends our Delaunay [2] and Delaunay$^{\mathrm{MM}}$ [1] systems. We therefore name our system *Delaunay–Coordinate a View* or *Delaunay*$^{\texttt{View}}$, for short. Delaunay$^{\texttt{View}}$ shares with Delaunay the capability to customize an individual view (an individual view can be a bar chart, a bipartite graph, a tree, or any other visualization defined by the user) and with Delaunay$^{\mathrm{MM}}$ the retrieval and visualization of multimedia data. However, as it will become apparent from the remainder of the paper, the system architecture, the actual prototype, and the design objectives are new. For example in Delaunay, the user starts with a database and defines for each of its data classes a template with which to visualize that class, using an alphabet of primitive symbols (such as circles, lines, text, etc.) and layout constraints. Delaunay$^{\texttt{View}}$ starts from where Delaunay left off. First, we assume that templates have already been defined using Delaunay but are not attached to specific objects, but to generic data objects. It is up to the user to bind those templates with whichever data objects need to be visualized, a capability that is not available in Delaunay. The other very important difference is that Delaunay did not support multiple coordinated views of heterogeneous databases. Delaunay$^{\mathrm{MM}}$ addressed heterogeneous multimedia databases, but its architecture differs substantially from that of Delaunay$^{\texttt{View}}$ in that it lacks its layered architecture and the capability to coordinate views.

The Delaunay$^{\texttt{View}}$ architecture supports three components according to their major functionalities:

- The *data processing component* that establishes semantic relationships among heterogeneous databases. This component allows an information expert to build a global schema out of the individual schemas from the various information sources and uses the RQL [4] query language to retrieve the data from the global schema. The complete description of this component is outside the scope of this paper, but brief descriptions will be given to demonstrate its capabilities.
- The *view building component* that enables users to create visualizations based on their preferences, applying an available or user-defined template (e.g., bar chart, bipartite graph, sorter). This component also allows the user to customize the visualization (e.g., by rotating the visualization).
- The *view integration component* that enables users to construct an integrated visualization consisting of several views defined using the previous component. The relative positions of the views on a screen and the coordination between these views are defined dynamically resulting in a single coordinated view.

As the data flows through the above components, it goes through a variety of transformations, becoming encapsu-

lated inside of metadata that describes the visual attributes that are added. This metadata determines: 1) the visualization of the data using a particular template; 2) the visual relationships that enable dynamic interactions within a coordinated view. We identify these different transformations and the corresponding data abstraction layers, and establish a standardized format for the metadata.

To describe the data as it flows through the layers, we use *data descriptors*. The semantics of the data is encapsulated in the data descriptors. We use XML, the current accepted format for data interchange, to define the contents of the data descriptors. Data descriptors facilitate the communication between adjacent layers in our proposed architecture.

The architecture of Delaunay$^{\texttt{View}}$ establishes a generic scheme to transform data and data relationships into views and coordinated views. In Delaunay$^{\texttt{View}}$, the representation and the visualization of the data are loosely coupled in two ways: (1) templates can be applied to any data sets; and (2) a generic message event service establishes the coordination between views.

The rest of the paper is organized as follows. In Section 2 we present related research. Section 3 gives an example that demonstrates our approach using the prototype we have built. The layered approach we have designed is described in Section 4. Section 5 describes the implementation of the various components of our prototype. Finally, in Section 6, we summarize our contributions and point to future work.

2. Related Work

The subject of coordinated visualizations as associated with data exploration has received considerable attention in recent years [7, 10, 8]. One of the concerns has been on how to allow for end users to create visualizations quickly and interactively. We review some of the approaches that are more closely related to our proposal. Two of them in particular, Visage [10, 5] and the Snap-Together Visualization [7, 6] have made major contributions to this research area.

Visage [10, 5] is a software environment prototype that consists of several components for supporting dynamic visualization generation and interactive information manipulation for information-intensive applications. It is an information-centric approach to user interface design that enables rapid generation of visualizations that is integrated from diverse sources based on the user's selections. It supports the integration of various visual displays in a flexible environment. Visage leverages the complementary features of different customized visualization and analysis tools in a coordinated way.

The Snap-Together Visualization (or Snap for short) [7, 6] is a software system that has a lightweight mechanism, an open architecture, and a generic user interface that allows for end users to rapidly and dynamically create customized and coordinated visualizations without programming. One of the most innovative features in Snap is the use of semantic relational database concepts such as key constraints, to model and enable the coordination between visualizations.

More recently, the InVision framework [8, 9] proposes an open, component based, and knowledge enabled software architecture for the rapid prototyping of information visualization solutions. It supports an architecture for the construction of coordinated views. The architecture aims at a generic solution for integrating various visual representations, each being chosen based on its suitability. Although this work appears to be still in the design stage, the creation of the InVision framework is a worthwhile effort to establish a common architecture for the design, implementation and integration of various visualization components, which addresses important issues in the future development of effective visualization solutions.

Our approach extends the previous approaches because of its layered architecture, the use of XML technologies with which we build an abstract representation for the data, views, layout, and coordinated views, the querying of heterogeneous data, and the loose coupling of the data representation and visualization.

The layered architecture and the use of XML ensures system interoperability, in that each system component can be implemented under different platforms using different programming languages with different techniques. For example, a particular data processing component can be replaced with any other data processing component that can convert the search results to our standard data schema as defined by the DTD. Likewise, the view building and view integration components need only comply with the DTDs that we have defined for the data in the corresponding layers.

The abstract data representation allows for the recording of the user's visualization progress through the XML data descriptors that contain the state of the data, views, layout, and coordination between views at any point of the visual interaction. This enables the storing of previous searches, views, and coordinated views, or to easily explore different possibilities within the same session by saving intermediate states and returning to them if desired.

Finally, our approach is based on the premise that data is fluid and dynamic and can originate from any source, therefore a general approach to the visualization of the data cannot consider only fixed formats or previously established data relationships. To address such requirements, we provide a mechanism to dynamically wrap heterogeneous data. The other approaches do not consider the existence of heterogeneous databases and therefore lack this capability. Using Delaunay$^{\texttt{View}}$, the interaction between views or within a coordinated view is established by data and data relationships represented by XML descriptors. An XML descriptor is obtained through querying, metadata, or user-defined data connections. When interacting with multiple views or coordinated views, there are queries that can be formulated, which do not require further access to the database, provided that the relevant data and metadata are part of the data descriptor or that new relationships can be established on the fly. Therefore, this mechanism is different from the mechanisms of Visage and Snap, which require data manipulations to an underlying database or data repository: relational queries in the case of Snap and scripts in the case Visage. Therefore, the way in which data is manipulated in Delaunay$^{\texttt{View}}$ further enhances the loose coupling capabilities between data and visualization already mentioned in Section 1.

3. Case Study

In this section, we present an example of the *Delaunay*$^{\texttt{View}}$ system as used to produce an integrated visualization of three related data sets. We chose a manufacturing application and in particular the visualization of a

bill of materials. A bill of materials is a list of parts or components required to build a product. In the screenshot of Figure 1 the manufacturing of commercial airplanes is being planned using a coordinated visualization composed of distinct individual visualizations, such as a bipartite graph, a bar chart, and a thumbnail sorter. The bipartite graph illustrates the part-subpart relationship between commercial aircrafts and their engines. The other components of this integrated display are: a bar chart, which can display any quantitative attribute (or aggregation of attributes), such as the number of engines currently available in the inventory of a plant or plants, and a thumbnail sorter of the maps associated with the manufacturing plants.

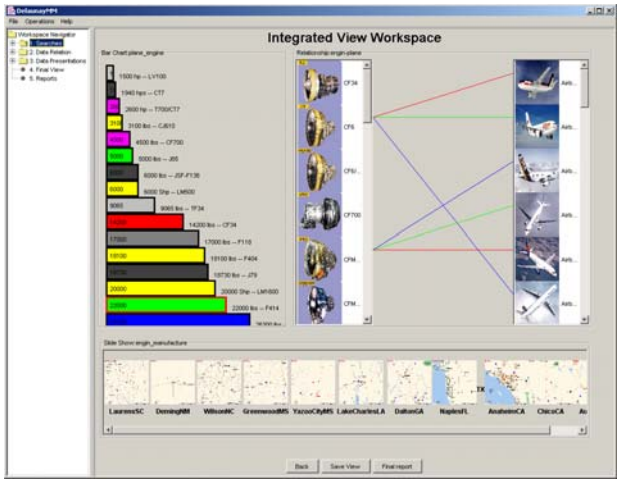

Figure 1. A coordinated integrated visualization.

The steps involved in building this visualization are:

1. The user constructs a keyword query to obtain a data set. This process may be repeated several times to get data sets related to airplanes, engines, and plants. The user can preview the data retrieved from query, make a refinement on the data, and name the data set for further use (the interface is not shown here).
2. Relationships are selected (if previously defined) or defined among the data sets, using either metadata, a query, or user annotations. In the former two cases, the user selects a relationship that was built by the data processing component (see Section 1). An example of such a relationship would be the connection that is established between the attribute *engine* of the airplane data set (containing one engine used in that airplane) and the engine data set. Other more complex relationships can be established by using an RQL query [4]. Yet another type of relationship can be a connection that is established by the user. This interface is shown in Figure 2. In this figure and those that follow, the left panel contains the overall navigation mechanism associated with the interface, allowing for any other step of the querying or visualization process to be undertaken. Note that we chose the bipartite component to provide visual feedback when defining binary relationships. This is the same component that is used for the display of bipartite graphs.

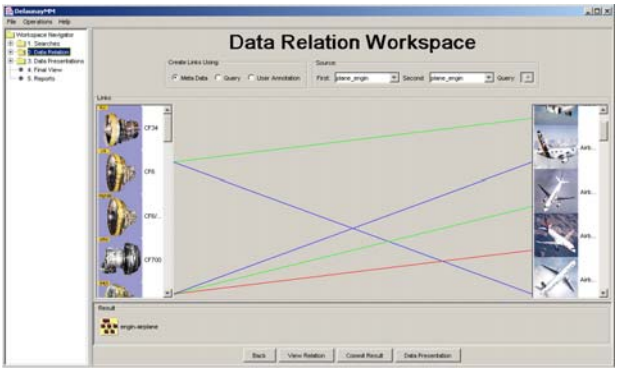

Figure 2. A relation workspace.

3. Individual views are built using templates. Users can apply different data sets to different templates to form different views. The interface of Figure 3 illustrates a thumbnail sorter of the maps where the manufacturers of aircraft engines are located. In this process, data attributes of the data set are bound to the visual attributes of the template. For example, the passenger capacity of a plane can be applied to the height of a bar chart. The users also can further change the view to conform to their preferences, for example, by changing the orientation of a bar chart. The sorter allows for the thumbnails to be sorted by the values of any of the attributes of the objects that are depicted by the thumbnails.

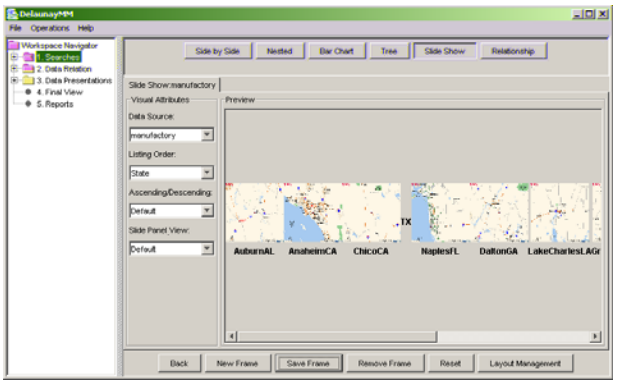

Figure 3. Construction of a view.

4. Individual views are laid out as in Figure 4, where we kept some space between the views, for clarity of the present description. The individual views can be placed anywhere on the panel. At this stage, the user selects from the panel that is immediately to the left of the composition panel the kind of dynamic interac-

tion between every pair of views that are being put together. Different views can be generated dynamically and be arranged in different places.

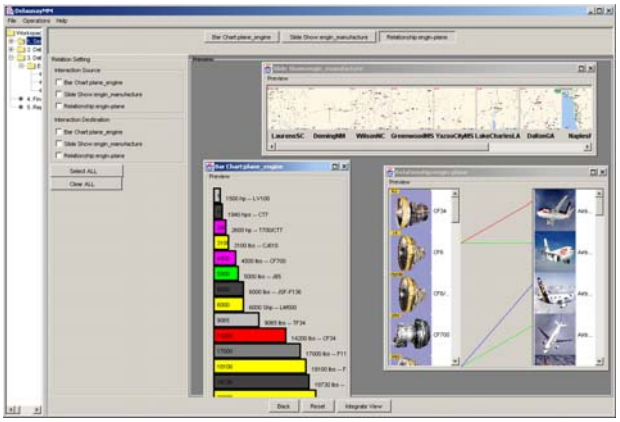

Figure 4. Layout views.

5. An integrated view is now built as represented earlier in Figure 1. In the integrated view, the coordination between individual views has been established. By selecting a manufacturing plant in the thumbnail sorter, the bar displays the inventory situation of the selected plant, for example, the number of available engines of each type. By selecting more plants in the sorter, the bar chart will display an aggregate number for the selected plants.

There are two ways of displaying relationships: they can be either represented within the same visualization (as in the bipartite graph of Figure 1) or as a dynamic relationship between two different views, as in the interaction between the bar char and sorter views. Other interactions are possible in our case study. For example, the bipartite graph can also react to the user selections on the sorter. As more selections of plants are performed on the sorter, different types of engines produced by the selected plants appear highlighted. Moreover, the bipartite graph view can be refreshed to display only the relationship between the corresponding selected items in the two data sets.

This simple case study illustrates several of the main characteristics of the *Delaunay*$^{\texttt{View}}$ system:

- Unique workspace, which supports all data selection and visualization definition mechanisms.
- Dynamic data set and relationship building, which are supported by queries or by metadata relationships.
- Flexibility of the visualization and of the coordination, which allows for different templates to be applied to the same data set and for the coordination relationships to be created by the user on the fly; the coordination between visualizations depends on the dynamic data set and relationship building process.
- Simple use of the visualization and integration components, which does not require programming from the end user.

4. Layered Approach

Layered architectures have been used in complex software systems to achieve a clean, scalable, flexible, and extensible design. By applying this software engineering paradigm in information visualization, we have discovered that not only the functionality of the system can be implemented in several layers but that also the data being visualized can be clearly described in such a layered format. As data flows through the different layers, information is added in each layer. We use the concept of *data descriptor* to define each data layer. A data descriptor should contain all the information obtained from that layer and from the layers below it. The higher the layer of the data descriptor, the more visualization related information it will contain. The simplified layered approach architecture is illustrated in Figure 5, which shows the corresponding data flow. Figure 6 illustrates the data descriptor contents and how they change through the layers.

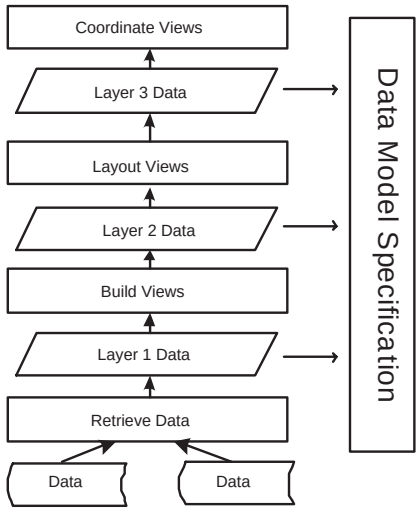

Figure 5. Layered data flow.

Data descriptors have all the information necessary for the system to build the views and coordinated views and are constructed using XML standard syntax. By using XML, we improve the capabilities of our system to communicate with other systems. The detailed functionality of the layers and their data descriptors are discussed in the rest of this section.

4.1. Data Retrieval

A simple common way for a user to compose a query is by specifying one “keyword”, a “source”, and some “criteria”. The underlying data retrieval engine will return to the user a data set satisfying the query with its associated metadata. A simple example of a data set as represented by a data descriptor is as follows:

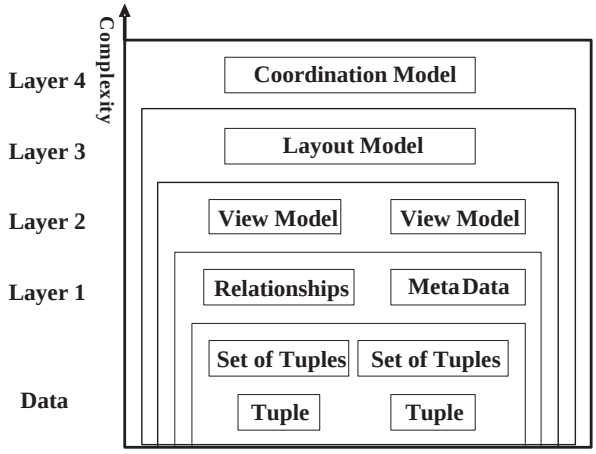

Figure 6. Layered data model.

```
<dataset name="engineData" type="relation"
    sid="set02">
        <query keyword="engine"
            source="inventory"
            criteria="commercial airplane"/>
                <metadata>
                <attr attr-
                    name="engineName"
                    type="text"
                    aid="a21"/>
                <attr
                    attr-name="Image"
                    type="image"
                    aid="a22"/>
                <attr
                    attr-name="Thrust"
                    type="numerical"
                    aid="a23"/>
                <attr attr-name="Type"
                    type="string"
                    aid="a24"/>
                </metadata>
                <tuple>
                <data attr-
                    ref="a21">GE90</data>
                <data attr-
                    ref="a22">Images/ge90.gif
                </data>
                <data attr-
                    ref="a23">115000
                    lbs</data>
                <data attr-
                    ref="a24">turbine</data>
                </tuple>
                <!– other tuples–>
</dataset>
```

A data set can be of type *relation* and *relationship*. A relation is just a set of tuples as in the relational data model. A relationship is also a set of tuples, but the attributes in the tuples are foreign keys of relations or are attributes of relations connected by a query. In the former case, the relationship is solely based on metadata knowledge about keys and foreign keys.

Each data set will have a unique user defined identity associated with it. The query element adds extra metadata, by specifying how the data was retrieved or a relationship formed. The metadata component contains the type and format of the data. The data component of the descriptor can contain actual data retrieved from the database or a link to the data content (e.g., to an image file).

4.2. View Building

As all the data is available in Layer 1, the user can interact with the system to build views to visualize the data sets in the descriptor. The system has a set of predefined visualization templates that can be set dynamically by the user to build a view of a data set. The properties of a visualization template belong to two categories: 1) Pairs containing the visual attributes (e.g., bar length in the case of a bar chart) and the data attribute they visualize (e.g., thrust in the case of an engine); 2) The attributes that are added by the user's customization of the templates (e.g., the orientation of a bar chart or the ordering of a sorter).

Upon finishing the interactive operations for building views using *Delaunay*$^{\texttt{View}}$, the system will construct a data descriptor in Layer 2:

```
<view template="barchart" dataset-ref="set02"
    name="barchart-engine" vid="v01">

        <view-property bar-length="a23"
            bar-color="a24"
            bar-label="a21"/>
        <customization direction="east"
            bar-ratio="50" bar-width="20"/>

</view>
```

In this example, the length of the bar is proportional to the thrust, its color encodes the engine type, and its label displays the engine name. Since more than one view can be built, there will be a group of such view templates appended to the data descriptor from Layer 1.

4.3. View Layouts

Using the data descriptor of Layer 2, the *Delaunay*$^{\texttt{View}}$ system can represent the view inside a floating window; the user can move and resize the window and place several windows in the same panel. The positions will be represented in Layer 3. An example of this data descriptor in this layer is as follows:

```
<layout name="scene1" width="400" height="600"
    lid="s1">

        <view-position view-ref="v01"
            width="200" height="400"
            xPos="0" yPos="0"/>
        <view-position view-ref="v02"
            width="200" height="400"
            xPos="200" yPos="0"/>
        <view-position view-ref="v03"
            width="600" height="200"
            xPos="0" yPos="400"/>

</layout>
```

Each view element in this layer must exist as a data descriptor in Layer 2, that is, it must have been defined by building a view.

4.4. Coordinated Views

The same set of views of the previous layer will be used to coordinate views. The user will set the possible coordination between views using the system provided interface. When a coordination occurs between two views, one is the *initiating view* and the other one is the *destination view*. The destination view will respond to certain types of actions on the initiating view (e.g., clicking on a visual element). For example, the visual representation of a particular object can change (e.g., be highlighted) if that object is related to the object whose visual representation was manipulated in the initiating view.

Once the coordination properties are defined by the user, the system can append metadata to the data descriptor of Layer 3, to form a data descriptor at Layer 4. An example of the new metadata to be appended is as follows:

```
<coordination layout-ref="s1"
    connection-type="one-to-many-aggregation"
    name="sorter-barchart" >
        <initiating-view view-ref="v02"
            selection-type="group"/>
        <destination-view view-ref="v01"
            reaction-type="update-model"/>
        <operation param1-ref="set03"
            param2-ref="set02"
            operator="sum" data-ref="ra1"/>
</coordination>
```

In this example, a sorter is the initiating view. The user selects a group of thumbnails in the sorter. As a result, the destination view, which is a bar chart representing for each engine the number of engines that are available, will change. The reaction type in this case is "update-model" meaning that an attribute in the view will represent a new attribute of the engine data set. In this case, previously the length attribute for a bar was related to the attribute *Thrust* for each engine. As a result of the aggregation, the length of the bar will display the available number of engines in all the selected plants. As more plants are selected, the aggregated quantity of the engines that are available in those plants will change, thus changing the length of the bars. This particular visualization refers to a relationship with id "ra1", which was established by metadata, connecting the plant and the engines that it manufactures. This relationship can be established with the interface of Figure 2 without using a separate query processor. Furthermore, the aggregation can be computed just by accessing the data descriptor.

5. Implementation

5.1. Data Driven Mechanism

In the Delaunay`View` framework, a user's progress through the visualization process is associated with the underlying XML descriptor. The system provides an environment that allows for the user to access underlying data and customize the visualization properties in such a way that data and visualization are persistent as an XML descriptor. So, the visualization process can be resumed from the saved descriptor. We describe here the data driven mechanism through which the internal visualization objects can be saved simultaneously and transparently into the XML descriptor, and vice versa. This bidirectional information binding of the XML data descriptor to the internal visualization and data objects requires several implementation components that are illustrated in Figure 7. In principle, the implementation of this data binding facility should be based on the DTD of the descriptor that defines the XML data descriptor.

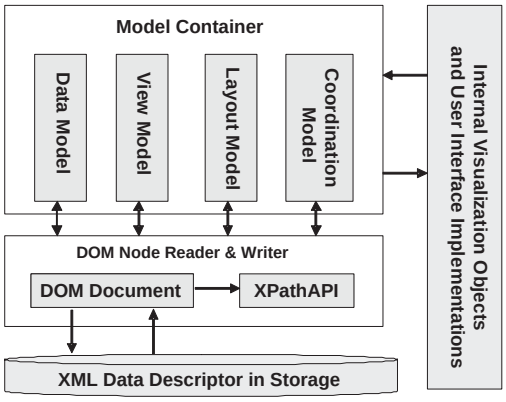

Figure 7. Architecture for the XML data driven mechanism.

The main component in this facility is the *model container*. Since the data descriptor has four layers, the model container also needs four types of *model classes* to provide different implementations of the *set model* and *get model* functionalities. A set model method will transform the system internal data structure of one of the data models into an XML DOM representation based on the XML data descriptor specification. A get model method will do the opposite, by creating the internal data structure from the DOM document. Consequently, data objects or visualization objects can be built based on that data structure.

The *DOM node data reader & writer* component uses the model class types and the information contained in the attributes to find the traversal path to the correct XML elements or attributes and to perform the data assignment or data retrieval. In particular, we use the XPathAPI class from Apache Xalan to simplify the implementation coding for the retrieval. By applying a document and Xpath expression to one of the static methods in XPathAPI, the method will return a node or a set of nodes that corresponds to the given Xpath expression in the document. An XML parser is used by the *DOM node data reader & writer* to read or write XML data from external storage. For a large XML data descriptor, the data manipulation adapter provides a cache mechanism to store part of the DOM tree in secondary storage. This XML data driven mechanism is utilized in all different stages associated with building an integrated visualization.

5.2. Building Views from Templates

One of the major functionalities in the Delaunay`View` framework is the possibility of building customized views by applying data to a view template.

A view template can be applied to any data set, and a data set can be visualized using any view templates. This flexibility is achieved by applying the MVC design pattern.

The *MVC (Model-View-Controller)* design pattern has been proved very useful in building visual interfaces (see, for example, [6, 8]). In our work, the model (Model) that is applied to the view template (View) is a subset of the data in Layer 1. In relational database terms, we could say that they result from a projection on some of the attributes, which will be visualized. By applying that subset, for example of engines (Model:Engines), to the view that is chosen by the user, for example sorter (View:Sorter), the system will build a concrete view (see Figure 3).

In the system implementation, a *view template class* has a corresponding *template model class* as the data provider. The template model class is the extension of model class with a specific type, which is tailored to a specific view template. So, the template model class can use the *XML reader & writer class*, which wraps a DOM document parsed from the descriptor, to get and set data on the fly. The template model class is based on the corresponding DTD that the descriptor conforms to. The view template classes implement the Layer 2 view customization features defined in the DTD.

5.3. Multiple Views Layout

The system provides an interface with which users can layout the views defined previously on a 2D panel, according to their preferences, by choosing their size and position. Such activities are similar to moving and resizing windows in most window software environments.

The users can continuously change the positions and views of the views. As soon as the user exits this process, the layout information will be exported as supplementary data information to the previous data descriptor (of Layer 2). The system will use the information of the layout model to present the integrated visualization through a container class. The container class consists of several view objects (built in previous stages) together with the (x, y) positions associated with the views, as provided by the layout model.

5.4. Coordinating Multiple Views

In order to establish coordinating multiple views, the system provides a user interface with which users select the interaction of the views with one another. Building this coordination relies on the information that was built in the previous steps through the data descriptors. The interaction between the views reflects the data relationships underlying the views that exist in the data descriptor of Layer 3. This coordination building facility supports:

- A generic way in which data, relationships, and views are built dynamically.
- A user interface for the specification of the properties of the coordination.
- Generic mechanisms that can dynamically (i.e., there is no static coding) establish interaction between views.

The architecture for establishing the coordination of views is based on a loosely coupled event notification mechanism that uses a mediator. The *mediator*, which is the core component, keeps track of the coordination information between views and establishes the interaction at run time. An *event* encapsulates data messages that need to be passed from one view to another. The interaction between views is communicated through the mediator in terms of an event. For example, if two views have an established coordination between them, when one action occurs in an initiating view (e.g., click on an object), an event will be forwarded to the destination view by the mediator, and the destination view will update its view upon receiving the event (e.g., highlighting of a related object). The major components and the different coordination events are illustrated in Figure 8.

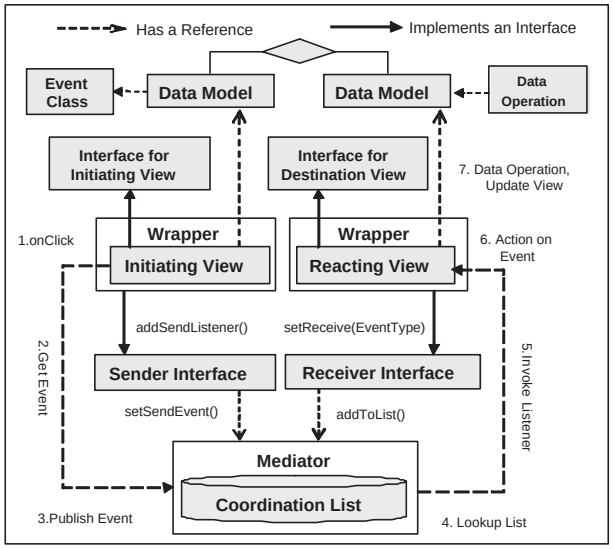

Figure 8. View coordination architecture.

Next, we describe in detail the different elements of the architecture:

1. The mediator keeps the information of which view can receive which event type and from which view that event can be received. This information is kept as a coordination list containing the initiating view identifier, the event type, and the destination view identifier. The mediator also has a programming interface that allows the view to register or publish the information associated with an event. The mediator is implemented as a singleton class [3]. At any time, the system only has one reference to it, therefore the mediator acts as a control center for dispatching the event.
2. The event classes are generated according to the data models that exist in the system. The connection between two views is based on the same type of event class. In order to connect two views dynamically, one of the views, the *initiating view*, uses the mediator to publish an event of its own data model type, while the other view, *the destination view*, has subscribed with the mediator to receive an event of that type. An event must have two attributes, one is the identifier for the initiating view and the other one is the generic data instance that contains data and metadata. When a coordinating action occurs, the initiating view will generate an event using its view identifier and related data set and will pass them to the mediator. The mediator checks the coordination list in order to inform the corresponding views of the event type and of the view

identifier. That is, the initiating view broadcasts the event, but the destination views are selected according to the events to which they subscribe to.

3. A set of interfaces is defined for the event notification services. The implementation of such an interface is mandatory for the view components whose coordination is established through the mediator. In our implementation, we have a wrapper class for the view component, which implements the common functionalities that are needed by the mediator. Such functionalities include calling the mediator to publish an event method using the event as the parameter.

   The wrapper class works as a container for an individual view component. It provides view components with an indirect way to use the mediator. Therefore, the view components do not implement the specific interface required by the mediator where the implementation code is the same for the different view components. The interface that view components need to implement will have different implementations depending on each view component.

4. The determination of a relation between two views needs to be conducted when a view component receives an event and an action is going to take place. A few relation operation functions are implemented as static methods thus becoming accessible to any view components in the implementation of the "actionOnEvent" method. The relation data passed to the functions are the objects of the generic data model that is presented in every view component. For example, if a linking coordination is established between two views, the receiving view component will obtain the data of the sending view from the event. The receiving component can use this data and the data associated with its own view as parameters to invoke the static method associated with the linking operation to find out the actual data item that the receiving view needs to be linked to.

6. Conclusions and Future Work

In this paper we presented a visualization system called Delaunay`View`. This system facilitates decision support applications by allowing users to explore heterogeneous data from multiple sources in a customized integrated view. Users can customize both the integrated view and the individual views that make up the integrated view.

The system features a layered architecture, such that data flows through the different layers and gets wrapped by metadata that describes the views and the interaction among the views.

The architecture for establishing the coordination of views is based on a loosely coupled event notification mechanism that uses a mediator to keep track of the coordination information between views and establishes the interaction at run time. The way in which the coordination is achieved is dynamic, that is, an interaction is possible between any two views independently of the way in which they were defined or of which data they display.

Future work will concentrate on the refinement of the layered architecture, on the exploration of interoperability issues, and on customization features. We intend to further refine the layered architecture so that we can have a complete history with rollback capabilities, that is, at any moment a session can be interrupted and resumed at exactly the same state where it was left off. This kind of feature is interesting from a usability viewpoint. We aim at completely characterizing the state of such a complex visualization system by enabling it to be captured by a data descriptor.

From an interoperability viewpoint, we would like to experiment with the implementation of the different layers using different programming languages and distributed platforms communicating with a web services architecture.

Finally, we would like to explore a semantic-based approach to customize the interface so as to reflect the preferences of a particular user or groups of users. Such an approach would annotate the connections among user interface components to reflect the users' choices in displaying a particular kind of objects or in following a certain sequence of actions.

Acknowledgments

This research was supported in part by the National Science Foundation under Awards ITR IIS-0326284 and EIA-0091489. We would like to thank Haiyan Lin for helping with the implementation.

References

[1] I. F. Cruz and K. M. James. A User Interface for Distributed Multimedia Database Querying with Mediator Supported Refinement. In *International Database Engineering and Applications Symposium (IDEAS '99)*, pages 433–441. IEEE Press, 1999.

[2] I. F. Cruz and P. S. Leveille. Implementation of a Constraint-based Visualization System. In *IEEE Symposium on Visual Languages (VL '00)*, pages 13–20, 2000.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Proceedings of the Eleventh International Conference on World Wide Web*, pages 592–603. ACM Press, 2002.

[5] J. Kolojejchick, S. F. Roth, and P. Lucas. Information Appliances and Tools in Visage. *IEEE Computer Graphics and Applications*, 17(4), 1997.

[6] C. North. *A User Interface for Coordinating Visualizations Based on Relational Schemata: Snap-Together Visualization.* PhD thesis, Computer Science Dept., University of Maryland., 2000.

[7] C. North and B. Shneiderman. Snap-Together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata. In *Working Conference on Advanced Visual Interfaces*, pages 128–135, 2000.

[8] T. Pattison and M. Phillips. View coordination architecture for information visualisation. In *Australian Symposium on Information Visualisation*, volume 9, pages 165–169, 2001.

[9] T. Pattison, R. Vernik, D. Goodburn, and M. Phillips. Rapid assembly and deployment domain visualisation solutions. In *Australian Symposium on Information Visualisation*, volume 9, 2001.

[10] S. F. Roth, P. Lucas, J. A. Senn, C. C. Gomberg, M. B. Burks, P. J. Stroffolino, J. A. Kolojejchick, and C. Dunmire. Visage: A User Interface Environment for Exploring Information. In *Information Visualization*, pages 3–12, 1996.