End-User Visualization and Manipulation of Aggregate Data

Authors: T. Paul McCartney and Kenneth J. Goldman

Aggregate visualization and manipulation enables the viewing and interaction of dynamically changing data sets in a graphically meaningful way. However, off-the-shelf applications generally provide only limited ways to view aggregates. To be truly effective to the end-user, an aggregate visualization should be customizable to suit the individual's needs. This paper describes a software system that empowers end-users to create interactive aggregate visualizations through direct manipulation. Included are mechanisms for specifying how aggregate data is processed from multiple sources, providing functionality similar to project, select, join, and cross product of relational databases. Visualization of distributed data sets is emphasized.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research
Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

End-User Visualization and Manipulation of Aggregate Data

T. Paul McCartney, Kenneth J. Goldman

WUCS-97-08

February 1997

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899
End-User Visualization and Manipulation of Aggregate Data

T. Paul McCartney, Kenneth J. Goldman
Department of Computer Science
Washington University
St. Louis, Missouri 63130, USA
{paul | kjg} @ cs.wustl.edu
http://www.cs.wustl.edu/cs/playground/euphoria/

Abstract

Aggregate visualization and manipulation enables the viewing and interaction of dynamically changing data sets in a graphically meaningful way. However, off-the-shelf applications generally provide only limited ways to view aggregates. To be truly effective to the end-user, an aggregate visualization should be customizable to suit the individual’s needs. This paper describes a software system that empowers end-users to create interactive aggregate visualizations through direct manipulation. Included are mechanisms for specifying how aggregate data is processed from multiple sources, providing functionality similar to project, select, join, and cross product of relational databases. Visualization of distributed data sets is emphasized.

Keywords: aggregate data, constraints, direct manipulation, distributed computing, matching, multi-way constraints, user interface management system, visualization

1. Introduction

Most large scale applications use aggregates to store collections of element data. For example, an air traffic control center may have a database that stores an aggregate of airplane information (e.g., identification, aircraft type, passenger/cargo classification, etc.); each airplane record represents an element of the aggregate. Visualization of an aggregate can make the information easier to comprehend, and may give the end-user insight that may not have been achieved otherwise. Direct manipulation with the visualization gives the user the ability to modify the data in a natural way. However, insight can only be achieved if the visualization reflects what the end-user would like to see. A visualization that includes extraneous information or uses an unintuitive graphical representation can be more confusing than viewing the aggregate information textually. Since each end-user may be interested in a different aspect of an aggregate, customization is the key to making an aggregate’s visualization effective to individuals.

Communications networks, such as the Internet, allow distributed applications to be constructed from communicating components (e.g., processes) running on separate computers. Parallel processing and support for multi-user applications are some of the benefits of a distributed approach. Visualization and manipulation of a distributed application’s data is particularly challenging. Many times, aggregate data is spread across multiple components of a distributed application. For example, the server component of a client-server application commonly stores an aggregate of information obtained from each client component. The application’s visualization component may be completely separate from the other components, requiring sophisticated communication and synchronization among the visualization and other components.
End-User Visualization and Manipulation of Aggregate Data

This paper discusses mechanisms that enable end-users to create customized, interactive aggregate visualizations. These mechanisms were developed in the context of state-based distributed computing, but are equally applicable to non-distributed applications. End-users can visually specify how aggregate data is processed from multiple aggregates, using mechanisms functionally similar to select, project, join, and cross product of relational databases [1], [2]. This allows aggregate data to be combined from different sources without the need for textual programming. End-users also visually create “mappings” of each aggregate to the graphics display, enabling both the visualization and manipulation behavior of the aggregates to be specified without programming.

The organization of this paper is as follows. Section 2 provides an overview of work related to aggregate visualization and manipulation. Background information about this work and the implementation environment in which it was developed is provided in Section 3. Section 4 describes aggregate mappings, the end-user mechanism for creating interactive aggregate visualizations. Section 5 discusses possible future research directions, and Section 6 ends with a brief summary.

2. Related Work

Commercial products such as Microsoft Excel [16] and LabView [8] provide some support for end-user aggregate visualization. Excel users may choose from a number of pre-defined chart representations (e.g., pie charts, histograms) to view spreadsheet data. Excel’s visualization is static; once a chart is created the user cannot interact with the display. Our system supports end-user interaction with the aggregate visualization and automatic updates of the display in response to dynamically changing data sets. The LabView visual programming environment allows users to construct array types interactively and, like Excel, allows plotting of the array data using built-in chart displays. Although LabView’s chart displays are updated when the underlying data is changed, aggregate data cannot be manipulated interactively.

Excel and LabView allow only limited customization of how the data is visualized. For example, it is possible to specify the color of a bar chart’s graphics in Excel. However, more involved customization such as creating a completely new type of chart representation is not supported in Excel or LabView. Our system gives end-users the ability to specify arbitrary graphical representations of aggregate data and how it is “mapped” to the display. This gives users the ability to create highly customized, interactive visualizations.

Graphics toolkits such as Garnet [16] and Interviews [9] provide programmatic support for visualizing data sets such as lists and graphs. Our work focuses on allowing end-users to create aggregate visualizations without textual programming. Research systems such as Pavane [17] and Weasel [3] allow the visualization of aggregate data through the use of a high-level specification language. Swarm, Pavane's specification language, allows programmers to write a formal textual specification how data from concurrent algorithms is to be mapped to a display. This specification is translated into a standard programming language which is compiled into an executable. Pavane visualizations cannot be manipulated. Our system allows end-users to specify aggregate visualization graphically at run-time. Visualizations are created which reflect the state of distributed applications in real-time, allowing the viewer to manipulate the data and change the behavior of the application as it is running. GVL provides a functional specification language for Weasel, and is used to specify the mapping from the program’s data state to the display. Like our work, Weasel decouples the program state from its visualization. Weasel provides support for visualizing dynamically changing data in real time, but does not support manipulation of the visualization. However, it is not clear that end-users (i.e., non-programmers) would be able to construct visualizations with Weasel.
3. Background

This section describes the software systems that this work was built upon, including a distributed programming environment and a user interface management system.

3.1 The Programmers' Playground

The Programmers' Playground [7] is a software library and run-time system that supports a new programming model for distributed applications. The model, called I/O abstraction, provides a separation of computation from communication that is well-suited for end-user construction of customized distributed applications from computational building blocks. Playground users do not need to write any source code to establish communication between the components of a distributed application, nor do they need to understand the details of how communication occurs.

In the I/O abstraction model, each module (i.e., process) of a distributed application has a data boundary containing published variables that may be externally observed and/or changed. Modules are written in a standard programming language (e.g., C++) using the Playground library. This library provides publishable data types, including base types (e.g., integer, real, string), tuples, and aggregates (e.g., arrays, lists). These types may be arbitrarily nested to form new publishable tuple types, and new publishable aggregates may be defined as well. Playground modules have a visual representation that was designed as part of a simple visual language for interprocess and intraprocess communication [13].

A distributed application consists of a collection of independent modules and a configuration of logical connections among the modules' published variables. Whenever a module updates one of its own published data items, the new value is implicitly communicated to all connected variables in other modules. The details of how the communication is handled are hidden from the implementor and users of the module. This simplifies module construction and gives the run-time system the ability to optimize communication. The configuration of connections is determined dynamically at run-time, rather than statically at compile time. End-users can use the module visual language to configure communication among Playground modules, giving the flexibility to add new components or relationships to their applications dynamically.

An element-to-aggregate connection between published variables results when a logical connection is formed between a data item of type T and an aggregate data item with element type T. For example, a client-server application could be constructed by having the server publish a data structure of type list(T) and having each client publish a data structure of type T. Element-to-aggregate connections also facilitate the development of distributed applications that gather aggregate data from many different sources.

3.2 EUPHORIA User Interface Management System

EUPHORIA, Playground's user interface management system [12], [14], is a specialized module for creating customized direct manipulation GUIs without the need to write user interface source code. In EUPHORIA, end-users simply draw GUIs using an interactive graphics editor (Figure 1). GUIs can consist of simple shapes, end-user defined widgets (encapsulated grouping of shapes), and images. Each shape drawn in EUPHORIA has a number of associated attributes (e.g., position, size, etc.) that can be used in forming relationships to other shapes and to external modules of a distributed application.
End-users can define multi-way constraints [12] among shape attributes, allowing graphical interaction behavior to be defined without programming. In EUPHORIA, the “handles” of a selected graphics object act as data ports to the object’s underlying attributes, allowing end-users to define constraints between graphics objects by drawing connection lines between their handles [14]. For example, one could constrain a rectangle object to be a square by drawing a connection line between its width and height handles, constraining its width and height attributes to be equal. Since constraints are multi-way, changing either the rectangle’s width or height causes the other to also change.

Certain shape attributes can be exposed to EUPHORIA’s data boundary as Playground published variables (see Figure 1). This allows other modules of the Playground system to selectively view or control the state of EUPHORIA’s display. Also, this decouples the application from the visualization since the application only operates in terms of its published state rather than the specifics of the graphical display. End-users can interactively construct and publish new tuple and aggregate data types.

![Figure 1: Interactive Orbital Simulation in EUPHORIA.](image)

Figure 1 shows the end-user constructed GUI for a distributed application that simulates the orbits of the Earth and the Moon around the Sun. This application consists of three Playground modules, including the EUPHORIA GUI. Graphical controls in the user interface allow the user to change the simulation properties, such as speed and rotation direction, through direct manipulation. The path of the Earth or Moon can be changed by moving any of three orbit representative points. End-user defined multi-way constraints are used to establish relationships among graphics shapes. For example, the orbit points controller consists of several circles and lines that are connected by constraint relationships. Moving any of the points moves the corresponding line end-point(s). Published variables are used to expose certain graphics attributes to external modules of a distributed application. For example, the center position of the

---

1. These three points specify the shape of an ellipse. Recall that the sum of the distances from the two foci of an ellipse to any point on the ellipse is constant.
Earth's picture is published as variable \textit{epos}. When an external change is received for \textit{epos} (i.e., communicated through a logical connection), the system responds by animating the Earth's picture to the appropriate location. Similarly, manipulating graphics items associated with EUPHORIA's published variables results in communication of the new state information to external modules.

A usability study of EUPHORIA was conducted on the Washington University campus. This study showed that end-users with no prior experience with distributed computing or user interface construction could learn to use EUPHORIA in a fairly short period of time (e.g., 30 minutes), and could construct interactive graphical user interfaces [11]. Subjects from a variety of backgrounds were chosen (i.e., not just computer science students). The usability study did not involve aggregate mappings explicitly, but the steps to create ordinary constraints within EUPHORIA are similar to the steps used to define aggregate mappings.

The above background discussion of The Programmers' Playground and EUPHORIA is sufficient for understanding the research contributions presented in this paper. Further details on The Programmers' Playground and EUPHORIA are discussed in [7], [12], [14].

4. Aggregate Mappings

The central contribution of this work is the \textit{aggregate mapping}, a mechanism that allows end-users to specify how aggregates are to be visualized. Aggregate mappings are a general-purpose approach to both creating the visualization and defining communication between aggregate data structures and their corresponding visualizations. Aggregate visualizations constructed in this way are dynamic, implicitly responding to dynamically changing aggregate data, and are interactive, reacting to direct manipulation by the user with the visualization. In The Programmers' Playground, this means that the external modules can drive the animation behavior of an aggregate visualization and can view changes to the aggregate's data that occurs from user interaction with the visualization.

The following subsections describe aggregate mapping mechanisms in detail. Section 4.1 presents an example application that uses aggregate mappings to visualize and manipulate aggregate data. Section 4.2 describes how end-users can define aggregate mappings interactively. Section 4.3 discusses end-user mechanisms for filtering aggregate data that is displayed. Section 4.4 presents end-user mechanisms for combining aggregate data from multiple sources through matching data fields.

4.1 Aggregate Mapping Example

Image morphing is the process of transforming one image into another, forming a series of intermediate images that animates the metamorphosis (see Figure 2). Morphing is commonly used in the entertainment industry to produce special effects for movies and television. In order to define the morphing operation, it is necessary to identify corresponding features on the two images (e.g., locating a person's nose in the "start" and "finish" images). A user interface for specifying these features is constructed in EUPHORIA, using aggregate mappings to identify the features. Given this specification, external modules can execute the morphing operation. Features are identified by arranging numbered markers over the images (e.g., in Figure 2, markers numbered "7," "8," "9" of the start and finish images correspond to the nose location).

In this example, the numbered tags over the start and finish images represent two aggregate mappings (one for each image). These mappings visualize arrays of tuples containing coordinate and identifier information (tag positions and numbers). No programming was involved in defining this visualization; it was created interactively. End-users manipulate the array coordinates by dragging the tags to the
Figure 2: Image morphing in EUPHORIA using aggregate mappings to identify facial features.

appropriate locations. This array information is stored in published Playground published variables. This means that external modules can view the array coordinate information and can use it to determine how to perform the morphing operation.

4.2 End-User Specification of Aggregate Mappings

Aggregate mappings are defined by specifying relationships among the aggregate’s element type and a prototype instance of the visualization. In EUPHORIA's data boundary (see Section 3.2), the element type is displayed as part of the published aggregate variable. For instance, the morphing example in Section 4.1 uses arrays of “Feature” tuples containing coordinate and identifier information; the element type of these aggregates is a Feature tuple (see Figure 3a). The prototype instance is a graphical representation (e.g., a simple shape or widget) that will be used to display each aggregate element. In the morphing example, the prototype instance is a single “feature marker” (i.e., a square widget containing a numerical display). As described in [14], widgets are constructed interactively by the end-user in EUPHORIA.

Recall that graphics object handles are used for forming constraint relationships in EUPHORIA (Section 3.2). Handles are also used in defining the equality relationships between the aggregate’s element type and the prototype instance. This is done by simply drawing connection lines between the prototype instance’s handles and the fields of the element type. For example, Figure 3b shows the relationships for the morphing application. The prototype instance’s center handle is constrained to be equal to the pos field (position) of the element type; the prototype instance’s text handle is constrained to be equal to the id field of the element type.

For each aggregate element, the system creates a copy of the prototype instance and inserts it into the display. The attributes of the copy are constrained to the corresponding element fields according to the
specified relationships between the element type and the prototype instance. The result is a collection of graphics instances whose attributes are associated with the aggregate’s elements. In the morphing example, a feature marker tag is created for each aggregate element. The id and pos fields of each element are used to set the attributes of the feature marker. For example, the id field is constrained to the text handle, meaning that the text of the feature marker displays the value of the id field. Since multi-way constraints are used, consistency between the aggregate and its visualization are automatically enforced. External updates to the aggregate are propagated to visualization instances through constraints, resulting in display updates or animation (Figure 4). Similarly, user direct manipulation with the visualization instances are propagated to the aggregate. In The Programmers’ Playground, this results in the updated state being sent out to external modules.

The aggregate mapping operation is functionally similar to the “project” operation in relational databases since only a subset of the element type’s fields need to be used in the visualization; only certain information is projected from the underlying data to the visualization.

4.3 Filtered Aggregate Mappings

It is often advantageous to view only a subset of an aggregate’s elements. Filtering of extraneous elements results in a faster display that is easier to comprehend. For this reason, we have developed a mechanism that allows the end-user to filter an aggregate mapping’s displayed elements based on a predicate. For example, one could specify the predicate of “id < 30” on Figure 3b’s aggregate mapping, having the effect of only visualizing feature markers whose ID number is less than 30. This filtering operation is functionally similar to the “select” operation of relational databases.
4.4 Joined Aggregate Mappings

Many times, single aggregates do not contain all of the relevant information needed to make a desired aggregate mapping visualization. Instead, information may be spread among multiple aggregates and, in the Programmers' Playground, modules of a distributed application. This is especially true of an application that is constructed from different components that were created by multiple programmers. With a *joined aggregate mapping*, the data of multiple aggregates is coordinated within an aggregate mapping based on matching operations.

As an example, consider the visualization of a graph structure. A common way to store edge information is in ordered pairs of reference IDs to vertices (Figure 5). This approach reduces redundant storage since vertex information (e.g., position) is only stored once in the vertex rather than in every incident edge. Also, this allows for a more concise representation of edges for cross product relationships; one edge ordered pair could represent many actual edges (see Section 4.4.2). However, the edge ordered pairs alone do not provide sufficient information to visualize the edges since the positions of each edge's end points are not included. In this example, a joined aggregate mapping allows the information of the Vertices aggregate to be used within the Edges aggregate mapping via a matching operation with the Vertices id field.

![Figure 5: Using join to visualize graph edges.](image)

In EUPHORIA, end-users specify matching operations by connecting *key fields* of aggregate variables in the data boundary. Connecting key fields is achieved in a way that is similar to how aggregate mapping relationships are formed: by drawing connection lines between fields of the aggregates' element types. Connections are drawn from a key field in a *primary aggregate* to a key field in an *indexed aggregate*. This has the effect of creating a virtual representation of the indexed's element type in place of the primary's key field. For example, Figure 6 shows how an Edges aggregate (primary) is joined twice with a Vertices aggregate (indexed). Joining the v1 and v2 key fields of Edges to the id key field of Vertices creates virtual representations of the Vertex tuple within the Edges aggregate that can be used in establishing an aggregate mapping. This means that Edges' aggregate mapping can use the position information contained in the Vertex tuples to determine each edge's end points.

For each primary aggregate element, aggregate mapping instances are created based on matching key field values among the element and its associated indexed aggregates. A matching operation is performed on each joined field of the primary element; the cross product of the matches of each joined field is used to create the element's visualization instances. For example, if half of the vertices have id=0 and half have id=1, specifying an edge of (0, 1) would result in the visualization of a bipartite graph (i.e., an edge between each 0 vertex and each 1 vertex is created). In this way, joined aggregates are functionally similar to "join" and "cross product" operations of relational databases.
4.4.1 Joined Aggregate Example

The Gallager-Humblet-Spira distributed minimum spanning tree algorithm [6] can be visualized using aggregate mappings. In this application, each vertex of the underlying graph is an independently running module. The modules work together to find the minimum spanning tree of the complete graph according to the Euclidean distance among the vertices. Each vertex communicates with its adjacent vertices through message passing. A central message-forwarding module is used to collect message data and control the animation of message exchanges. Concurrently, tree fragments are formed by merging adjacent fragments, starting with single vertex fragments and ending with the minimum spanning tree.

In EUPHORIA, an interactive visualization of this algorithm can be constructed (Figure 7). Both the vertex and message visualizations are created through the use of aggregate mappings. Element-to-aggregate
connections (Section 3.1) are used to create the vertices aggregate by collecting vertex information from each individual vertex module. Since the edges are specified by the IDs of their end-points, the edges visualization is created through the use of a joined aggregate mapping with the vertex aggregate (see Section 4.4). As the algorithm proceeds, state information is visualized in EUPHORIA. For example, when a message is passed between vertices, a textual message object is animated between the vertices. The user can arrange the vertices through direct manipulation to specify any arbitrary graph topology.

4.4.2 Match Storage Algorithm

A joined aggregate’s matches may change incrementally over time in several different ways. For example, key field values of one or more primary and/or indexed aggregate elements may be changed at any time by its application. Also, values from primary and/or indexed aggregates may be added or removed at any time by either the application or the user. For reasonable performance, the storage for matches must be fast, must support cross products efficiently, must keep track of existing matches to avoid redundancy, and must be able to quickly remove matches in a variety of ways.

Consider the process of enumerating a single primary aggregate element matches. The matches can be viewed as an \( n+1 \) term cross product, where the first \( n \) terms represent the indexed aggregate element values matching each of the primary’s key fields and the last term represents the primary element. For example, Figure 8 shows the visualization of a graph using a two-level joined aggregate, as described in Section 4.4. Given \( \text{Edges} \) and \( \text{Vertices} \) aggregates, a cross product is formed using the matches of the first key field (i.e., index value 0), matches from the second key field (i.e., index value 1), and the edge value (i.e., \( \{0, 1\} \)). The specification of only a single edge results in a visualization of a bipartite graph, since multiple vertices match against the edge’s start and end points. That is, the aggregate mapping visualizes an edge between every vertex with \( \text{id}=0 \) to every vertex with \( \text{id}=1 \).

<table>
<thead>
<tr>
<th>Edges</th>
<th>( { (0,1) } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>( { (X,1), (A,0), (Y,1), (B,0), (Z,1) } )</td>
</tr>
<tr>
<td>Cross Product: ( { (A,0), (B,0) } \times { (X,1), (Y,1), (Z,1) } \times {0,1} )</td>
<td></td>
</tr>
</tbody>
</table>

**Visualization:**

![Figure 8: Example cross product for a bipartite graph and resulting visualization.](image)

The result of the cross product is a series of tuples of size \( n+1 \), each representing a match. Figure 9a shows an example cross product of a two-level join. The first key field (e.g., \( v_1 \) from Figure 6) matches against element values \( l_1 \) and \( l_2 \) of its indexed aggregate. The second key field (e.g., \( v_2 \) from Figure 6) matches against element values \( l_3, l_4, \) and \( l_5 \) of its indexed aggregate. The result is a series of six match 3-tuples. An \( n+2 \) level complete trie is used to maintain information about the current collection of match tuples. Each inner trie level represents the index aggregate matches to the key fields. Each path from the root to a leaf represents a match n-tuple. For example, Figure 9b shows a trie for an aggregate with two joined
fields. This example shows the first three match tuples of the cross product as well as the match tuples of other cross products. The highlighted path represents the \((I_1, I_4, P_1)\) match.

\[
\text{indexed #1} \times \text{indexed #2} \times \text{primary element} \\
\{l_1, l_2\} \times \{l_3, l_4, l_5\} \times P_1
\]

\[
\{l_1, l_5, P_1\} \\
\{l_1, l_6, P_1\} \\
\{l_1, l_9, P_1\} \\
\{l_2, l_9, P_1\} \\
\{l_2, l_4, P_1\} \\
\{l_2, l_6, P_1\}
\]

(a)

(b)

Figure 9: (a) Cross product of a primary element with its matches. (b) Storage trie for resulting matches.

This method of storage has a number of advantages. First, adding a new set of match tuples is efficient. It is not necessary to check the entire collection of match tuples to see if a new match tuple already exists. Instead, a new match tuple is simply weaved into the trie. Second, for large cross product operations, this storage representation is space efficient as compared to storing each match grouping separately as a list. Third, removing a single match term is efficient. For example, suppose that \(I_1\)'s key value is changed by the application. This means that it is necessary to remove all match tuples whose first term is \(I_1\). This is achieved by simply deleting the sub-tree rooted at the \(I_1\) node (and all associated graphical instance representations). Finally, deleting a graphic instance from an aggregate mapping is efficient. This delete operation involves deleting the nodes up the trie that are unique to the match tuple, starting at the appropriate leaf node.

5. Future Work

A number of enhancements are planned to the aggregate mapping mechanisms described in this paper. This section outlines some of these enhancements.

Currently, the only aggregate type supported in EUPHORIA is fixed size array\(^2\). EUPHORIA will eventually support other Playground aggregate types such as lists and mappings. Playground also has a general-purpose aggregate type called "grouping" [18] that can be used to construct other aggregate data structures such as trees and heaps. Since the structure of such aggregates is arbitrary, the end-user mechanism for specifying traversal order is an issue to consider (order will be important for aggregate layout, outlined below).

In the future, we would also like to extend the editing capabilities to an aggregate mapping. In the current implementation, an aggregate mapping's relationship between the prototype instance and the element type cannot be modified once the mapping has been achieved. Providing the ability to edit this relationship

\[\text{---}2. \text{End-users determine the array's size ahead of time. Once instantiated, the size is fixed.}---\]
would allow users to modify the constraints between the aggregate and its visualization instances on the fly. Also, end-users can currently delete visualization instances in EUPHORIA but cannot add new instances. The desired behavior in response to adding a new visualization instance would be to insert a new data element into the corresponding aggregate. The end-user mechanism for specifying the new element and how it should be added to the aggregate is still under consideration.

It would be helpful to have an end-user mechanism to specify the layout of aggregate mapping instances. This mechanism would enable a further decoupling of applications from their GUIs by eliminating the need for the application to specify element position information. One approach to this problem is to specify the layout graphically through induction-style rules. With this strategy, similar to a demonstrational approach [4], [15], the end-user would specify the relationship between the first aggregate element (base case), an aggregate instance k, and the relationship to the next element k+1. From this information, the system would inductively place each instance element of the aggregate. In addition, a number of boundary conditions could be included to deal with special cases. For example, consider the visualization of items in a row-column organized table. The position of each item is determined by the size of previous items and the dimensions of the table (i.e., for wrapping). With layout rules, the end-user could specify this organization in EUPHORIA without relying on external modules to explicitly set the position of each item.

6. Summary

Aggregate mapping is a mechanism that enables end-users to visualize and manipulate the data of an aggregate, such as an array or a list. No textual programming is required. End-users specify aggregate mapping visualizations through interaction with the EUPHORIA user interface management system. Used in conjunction with the Programmers' Playground distributed programming environment, aggregate mappings offer the end-user a way to customize distributed applications, enabling end-user construction of graphical user interfaces and visualizations.

By nature, a distributed application's state is divided among potentially many components. This property makes it difficult to construct visualizations since information must be combined from different sources in a meaningful way. We have presented several mechanisms to address this problem. Playground's element-to-aggregate connection type enables data to be aggregated from many distributed application components. Filtered aggregate mappings enable the end-user to selectively view only certain aggregate elements, resulting in a display that is easier to understand. Joined aggregate mappings provide a way to combine data from multiple aggregates into a single aggregate mapping, using an end-user defined matching. These mechanisms are powerful, providing functionality similar to that of project, select, join, and cross product of relational databases.

Acknowledgments

We thank David Saff for developing the module to perform the image morphing operation. This research was supported in part by National Science Foundation grants CCR-91-10029, CCR-94-12711, and ARPA contract DABT63-95-C-0083.

References


[17] Roman, G.-C., Kenneth Cox, C. D. Wilcox, and J. Y. Plun. Pavane: a system for declarative visualization of