EUPHORIA: End-User Construction of Direct Manipulation User Interfaces for Distributed Applications

T. Paul McCartney, Kenneth J. Goldman, and David E. Saff

WUCS-95-29

August 1995

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

EUPHORIA: End-User Construction of Direct Manipulation User Interfaces for Distributed Applications

T. Paul McCartney, Kenneth J. Goldman, and David E. Saff
Department of Computer Science
Washington University
St. Louis, Missouri 63130
{paul, kjg, des}@cs.wustl.edu
http://www.cs.wustl.edu/cs/playground/

ABSTRACT

The Programmers' Playground is a software library and run-time system for creating distributed multimedia applications from collections of reusable software modules. This paper presents the design and implementation of EUPHORIA*, Playground's user interface management system. Implemented as a Playground module, EUPHORIA allows end-users to create direct manipulation graphical user interfaces (GUIs) exclusively through the use of a graphics editor. No programming is required. At run-time, attributes of the GUI state can be exposed and connected to external Playground modules, allowing the user to visualize and directly manipulate state information in remote Playground modules. Features of EUPHORIA include real-time direct manipulation graphics, constraint-based editing and visualization, imaginary alignment objects, user-definable types, and user-definable widgets with alternative representations.*

KEYWORDS: constraints, direct manipulation, distributed applications, graphical user interfaces, multimedia, user interface management system

1 INTRODUCTION

Distributed multimedia applications supported by a global electronic infrastructure have tremendous potential for providing users with customized communication and computation environments. Applications include remote collaboration, information and resource sharing, and access to broadcast media. Future users of the infrastructure will vary greatly in technical ability, ranging from novice users to sophisticated expert users and programmers. Since communication and computation requirements vary by context and change dynamically, it is unlikely that off-the-shelf applications will anticipate the needs of all users. Therefore, empowering end-users to create their own customized applications for both communication and computation is an important challenge. Support for end-user construction of distributed applications means not only that users be able to combine software components, but also that they be able to construct user interfaces for interaction with these custom applications. Therefore, a complete solution to the end-user construction problem requires a user interface management system. Ideally, the user interface management system should integrate the tasks of distributed application configuration and graphical user interface construction.

The Programmers' Playground [7][8][9] 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 modules of a distributed application, nor do they need to understand the details of how communication occurs.

This paper describes EUPHORIA[1], a module of the Playground distributed programming environment that allows end-users to create direct manipulation graphical user interfaces exclusively through the use

[1]EUPHORIA is an acronym for End User Production of grapHical interfaces fOr Really Interactive distributed Applications.

of a graphics editor. Two different forms of communication occur as part of a complete, graphical distributed application. There is communication among the graphics objects within the user interface, and there is communication among the user interface and the other modules of the distributed application. EUPHORIA simplifies the application development process by carefully integrating the user interface construction process with the specification of the distributed application communication structure. End-users control both forms of communication, both forms may be changed dynamically at run-time, and neither requires any programming.

Within EUPHORIA, a direct manipulation GUI is created by end-users through the use of a graphical tool palette. Communication among simple and compound objects within the GUI is achieved by establishing constraints among the attributes of the graphics objects. Similarly, to complete the distributed application, the user establishes communication between the GUI and the other modules of the distributed system by creating logical connections between the GUI module data boundary and the data boundaries of the other modules. In this way, end-users can create a GUI completely independent of its application. An application can also have multiple customized GUIs (i.e., multiple users) each of which displays information in a different way.

The remainder of this paper is organized as follows. Section 2 discusses the theory and software tools of The Programmers' Playground. In Section 3, overview of related work is provided. Section 4 presents the EUPHORIA user interface management system followed by two example applications in Section 5. Section 6 presents an overview of the design of EUPHORIA. Section 7 discusses possible further research and Section 8 describes the current implementation status.

2 THE PROGRAMMERS' PLAYGROUND

EUPHORIA is implemented in the context of The Programmers' Playground. This section provides some background on Playground and the *I/O abstraction* model on which it is based. We limit this background discussion to concepts necessary for this paper. Details on Playground may be found elsewhere [9]. In the I/O abstraction model, each module in a distributed system has a *data boundary*[2] containing published variables that may be externally observed and/or manipulated. Modules are written in a standard programming language (e.g., C++) using the Playground library. The Playground library provides a set of publishable data types. These include base types (e.g., integer, real, string), aggregate types (e.g., arrays, mappings), and tuples. Programmers may arbitrarily nest these types to form new publishable data types, and new publishable aggregates may be defined as well.

A distributed application consists of a collection of independent modules and a configuration of *logical connections* among the published variables in the module data boundaries. 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 flexibility in optimizing communication. The configuration of connections is determined dynamically at run-time, rather than statically at compile time. This gives users the flexibility to add new components or relationships to their applications dynamically.

I/O abstraction communication is declarative, rather than imperative. That is, one declares high-level logical connections among the data items of individual modules, as opposed to directing communication within the control flow of the module. Output is implicit, a by-product of computation. Input is observed passively, or handled by reactive control within a module. This declarative approach simplifies application programming by cleanly separating computation from communication. Software modules

[2]In other papers describing the I/O abstraction concept, the data interface of an I/O abstraction module has been called the "presentation." Since this paper deals with user interfaces, we use the term "data boundary" in order to avoid confusion.

written using I/O abstraction do not make explicit requests to establish or effect communication, but instead are concerned only with the details of the local computation. Exposing the configuration also allows the run-time system to handle communication more effectively.

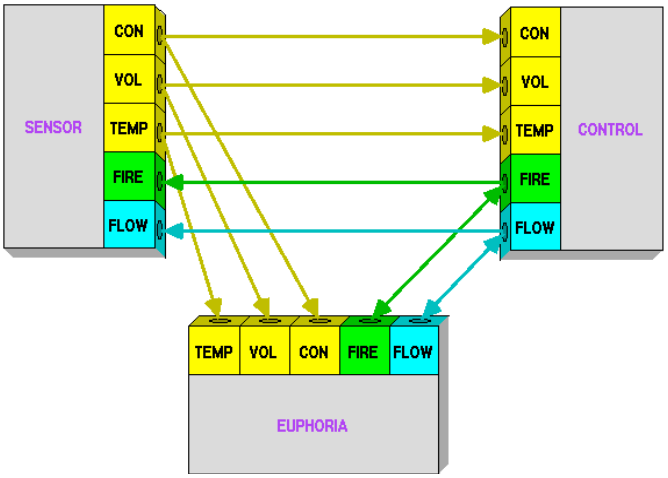

Figure 1: Playground modules with logical connections.

Playground modules and logical connections have a visual representation (Figure 1). A Playground module (i.e., an active process) is represented as a box with a data "plug" for each variable in the module's data boundary. The color of each variable represents its type. Logical connections are represented as arrows between pairs of variables in module data boundaries. The metaphor is that of wiring together the components of a stereo system, where the color of each cord denotes the type of information that it carries.

3 RELATED WORK

EUPHORIA's contributions impact two distinct areas, graphical coordination languages for configurable distributed systems and end-user construction of graphical user interfaces. In this section we highlight some of the related work in each of these areas.

The purpose of a *coordination language* [5] is to separate communication from computation in order to offer programmers a uniform communication abstraction that is independent of a particular programming language or operating system. The separation of computation from communication permits local reasoning about functional components in terms of well-defined interfaces and allows systems to be designed by assembling collections of individually verified components. Coordination languages typically provide a structured configuration mechanism for specifying relationships among program modules. For example, Darwin [18] is a configuration language for managing message-passing connections between process *ports* in a dynamic system. Processes are expressed in a separate computation language that allows ports to be declared for interconnection within Darwin. Conic, the predecessor of Darwin, provides a graphical configuration mechanism for establishing bindings among the ports [17]. However, the modules of the system must still be concerned with when to send or receive messages on these ports. In Polylith [25], a configuration is expressed using "module interconnection constructs" that establish procedure call bindings among modules in a distributed system. The Weaves system [10] provides a configuration mechanism based on dataflow.

The ViewStation system [27] provides support for interactive media-based applications, where modules perform explicit communication using *send* and *receive* primitives. The VuSystem programming environment includes a set of programming conventions, media processing elements and a TCL-based [24] GUI for specifying both in-band media communication and out-of-band control communication.

User interfaces to these applications are typically constructed by writing TCL scripts.

Extensive work has been done in the area of user interface construction. Thinglab [2] uses constraints to specify relationships between parts of a simulation graphical display. Thinglab represents early work in graphical constraint systems and provided the foundation for many later systems. Garnet [23] provides a toolkit which allows the user to construct interactive graphical user interfaces using an object oriented constraint-based library. Garnet does not provide end-users with graphical mechanisms for establishing relationships between the user interface and the application that it controls.

Fabrik [14][19] and LabVIEW [15] provide self-contained visual programming environments for sequential computation. Fabrik represents visual programs as data flow graphs of connected component icons. "Pins" are used as part of an overall electronics store metaphor, representing data ports used in connecting the components of a visual program. Fabrik's pin and component mechanisms are similar to the "plug and play" nature of Playground's module abstraction (Section 2) and end-user defined widgets (Section 4.8). LabVIEW is a commercial visual programming environment designed for use by engineers and scientists with little or no traditional programming experience. The basic component of any program in the LabVIEW environment is a "virtual instrument," that consists of a front panel and a block diagram. Programs are written in G, a data flow based language with a special set of additional control flow. Neither Fabrik nor LabVIEW support distributed applications. Both have a fixed set of predefined graphical components. In contrast, EUPHORIA supports end-user construction GUIs for distributed applications out of user-defined graphics components. The computational components of a distributed Playground application are created by programming in an existing language (e.g., C++). Our graphical tools are then used to define relationships among the states of modules in a distributed system and the state of the GUI, as well as relationships among graphics objects within the GUI.

The Rendezvous project [13] concentrates on the separation of user interfaces from their applications through the use of interprocess communication. Rendezvous is a transition from purely user interface oriented systems to systems that decouple the construction of the graphical user interface from their applications. GUIs are constructed by creating programs using MEL, a language extension to Common Lisp providing support for graphics operations, object-oriented programming, and constraints. Constraints are used as Rendezvous' interprocess communication mechanism between an *abstraction* (controlling source code) and a *view* (its visualization). Playground does not utilize constraints for interprocess communication. Instead, constraints are used exclusively to define relationships between the attributes of graphics objects within a GUI. Playground's interprocess communication abstraction, logical connections, decentralizes the communication of modules in a distributed system; Playground modules communicate asynchronously without the need for a centralized constraint solver.

4 EUPHORIA USER INTERFACE MANAGEMENT SYSTEM

The EUPHORIA user interface management system is based on a visual language for describing communication among modules in a distributed system and graphics components of a user interface [22]. This section describes how users create simple and compound objects of a user interface and define relationships among those objects. Relationships include constraints and encapsulation of graphics object groups by means of a data boundary.

EUPHORIA's graphics editor consists of three parts (see Figure 2). The body of the window is used for drawing and manipulation of GUIs. The top of the window contains a palette of drawing commands including basic graphics objects (e.g., rectangles, ovals) and user/system defined compound graphics objects (i.e., widgets). The left side of the editor window contains the data boundary portion of the editor. Both the tool palette and the data boundary may be hidden once construction of a GUI is completed.

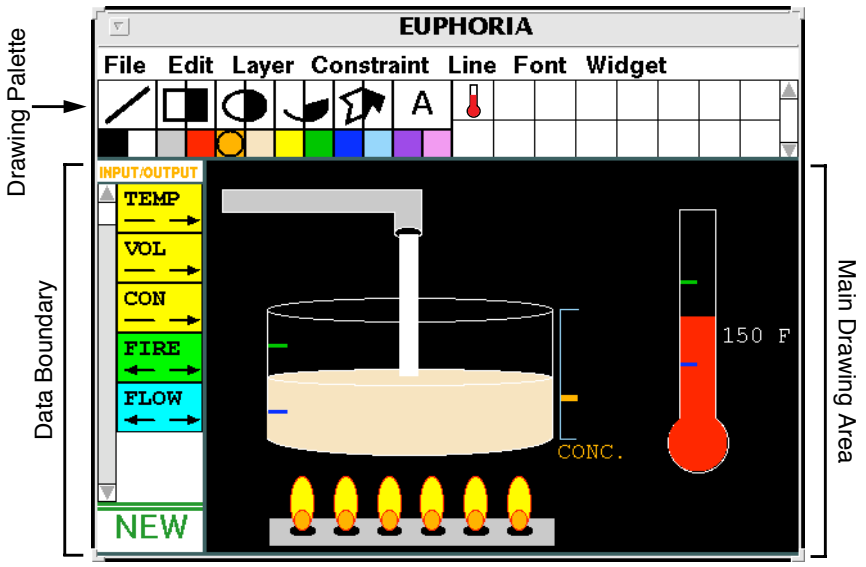

Figure 2: EUPHORIA graphics editor.

4.1 Graphics primitives and attributes

EUPHORIA supports a number of different types of graphics objects including rectangles, text, and images. Each graphics object has a set of *attributes* whose values define not only its visual appearance but also other state information such as whether or not it is "selected." Graphics object attributes are visually represented as handles that are positioned in appropriate places relative to the object. Handles serve a dual role in EUPHORIA. In addition to dragging, the user can also establish relationships (i.e., constraints) among the attributes of the graphics objects by using the handles as data ports.

4.2 Spaces

A *space* is a coordinate system that contains graphics objects. For example, the main drawing area of the EUPHORIA window is a space. To simplify user interface construction, EUPHORIA allows end-users to define multiple spaces with independent coordinate scaling factors, origins, and bounding rectangles. User defined coordinate systems allow a user interface to be defined in terms of meaningful values instead of raw pixels. In this way, external Playground modules need not be aware of how their GUIs display information.

4.3 Constraints

Constraints are a simple, yet powerful, way to specify persistent relationships among graphical objects. End-users can establish constraint relationships among graphics object attributes. Once a constraint is formed, the constraint solver is responsible for maintaining the relationship when changes are made to connected graphics objects or published variables.

Three types of constraints can be established by end-users: constant, equality, and conversion. *Constant constraints* are formed on an attribute of a graphics object by "anchoring" its corresponding handle. *Equality constraints* are formed by dragging a connection line between two graphics object handles. A *conversion constraint* is a specialized equality constraint in which the graphics attribute data types are compatible, but not the same, requiring a type conversion (e.g., connecting a real number graphics attribute to a string graphics attribute).

Figure 3 shows the process of inscribing an oval within a rectangle through the use of constraints. A constraint between the oval and rectangle left-top corners is formed by dragging a connection line between the corresponding handles. This action causes the oval to "snap" to the position of the rectan-

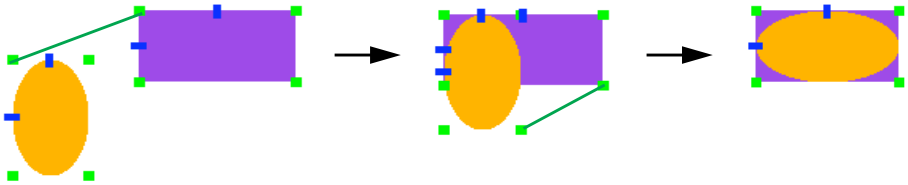

Figure 3: Inscribing an oval within a rectangle.

gle. A constraint is also formed between the right-bottom corners of the oval and the rectangle, inscribing the oval within the rectangle. These constraint relationships are maintained by the constraint solver; moving or resizing either the oval or the rectangle causes the other to change as well.

4.4 Constraint visualization

Since constraint graphs in EUPHORIA can grow large, the user may want to examine or edit the constraint graph to verify that it behaves in the expected manner. Constraints among selected graphics objects can be visualized and edited, with constraints to imaginary or off-screen objects hidden unless the user decides to display them. This allows for selective viewing of constraint information.

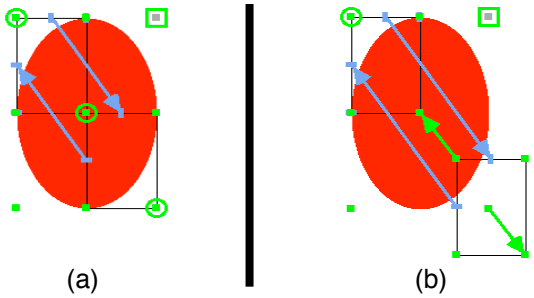

Figure 4: Constraint visualization in (a) normal view and (b) displacement mode.

An equality or conversion constraint is shown as a line between the handles that represent the values it constrains; constraints in which these handles are close are shown as circles around the handles (Figure 4a). An anchor is shown as an open square around the handle it anchors. All visualized constraints flash to distinguish them from the surrounding objects. The colors between which the constraints flash indicate the types of values constrained. The thickness of the constraint shows its hierarchical strength (see Section 6.3). Constraints not presently enforced are shown as broken lines. If the user chooses to display the propagation direction of the constraints, arrowheads point towards the values that were last changed by each constraint. Users can also change constraint strengths or delete constraints by interacting with the constraint visualization.

In certain cases, it may be difficult or impossible to determine to which objects a constraint is connected or the propagation direction of a constraint, especially when a number of objects are constrained at one corner (see Figure 4a). In these cases, the user may temporarily displace the apparent positions and sizes of objects on the screen. However, this editing has no effect on any other objects on-screen; the "real" positions and sizes remain as they were previously. This allows the user to "pull" graphics objects apart whose corners are constrained together, in order to view, edit, or delete the constraints between them (Figure 4b). When the editing is complete, the displacements are reset and the objects "snap back" to their original positions.

4.5 Imaginary objects

Many desirable constraint relationships cannot be established by equality constraints directly between

graphics object attributes. For this reason, EUPHORIA supports *imaginary objects*. Imaginary objects are invisible shapes that serve as an abstraction for defining indirect constraints between attributes. Any graphics primitive or widget can serve as an imaginary object. The attributes of an imaginary object can be constrained with the attributes of other graphics objects. In this way, users can create indirect constraints among graphics objects visually using the same mechanism used to create constraints among the attributes of visible objects.

For example, an oval in EUPHORIA does not have a handle for its center. However, through the use of two rectangles and a few constraints, it is possible to create an oval center handle (see Figure 4a). The sizes of the two rectangles are constrained to be equal and the right-bottom corner of one rectangle is constrained to be equal to the left-top of the other. These rectangles are then inscribed within the oval. The result is a handle that always remains in the center of the oval. The rectangles are then hidden by making them imaginary.

4.6 Data boundaries

The data boundary (see Figure 2) represents the subset of the attributes within the drawing area that are exposed to external Playground modules as published variables (see Section 2). When a published variable is changed in an external module, Playground sends the change out to all connected modules, including EUPHORIA. Similarly, when a graphics object is changed (e.g., moved by the user), this change may also be sent out to external Playground modules, according to the logical connections between published variables. Animated visualizations and interactive direct manipulation GUIs can be created by connecting appropriate attributes of a EUPHORIA drawing to external modules.

EUPHORIA supports all Playground base types (e.g., integer, string) and tuples. Each published variable is represented as a color-coded rectangle with a variable name and read/write permissions. Tuple variables containing a number of heterogeneous fields can be created interactively and viewed hierarchically [21].

4.7 Alternatives

A space can have multiple representations called *alternatives*. For example, a simulation GUI might consist of an alternative that shows the simulation state graphically, allowing direct manipulation, and an alternative that shows expanded information in a more "text and button" type representation. Alternatives are useful in the development of widgets.

4.8 Widgets

A widget is an encapsulated space containing graphics objects with a set of published attributes. Widgets are created visually by end-users. One may think of a widget as a "module" of graphics shapes with a "data boundary" of externally readable/writable attributes. The data boundary of a widget defines the subset of attributes which can be used externally by its container to control the widget's appearance. As with other graphics objects, the external attributes of a widget can be viewed, revealing handles that can be used for direct manipulation or in forming connections to the widget.

For example, the thermometer widget in Figure 2 can be constructed by an end-user as follows. First, the component shapes of the thermometer are drawn and constraints among the shapes are formed (Figure 5a). A scaling factor for the widget space is set using a coordinate system tool so that the top of the thermometer represents 300 degrees Fahrenheit. This allows external applications to interact with the thermometer in terms of real world values rather raw pixels. Second, the data boundary of the widget is defined by publishing the height of the mercury (i.e., the temperature, see Figure 5b). The data boundary specifies that only the temperature attribute will be exposed from the widget when it is used. The specification of the widget is saved, and the widget can then be used within a GUI (Figure 5c). Notice that the only handles on the selected thermometer are the default bounding box handles and the temperature

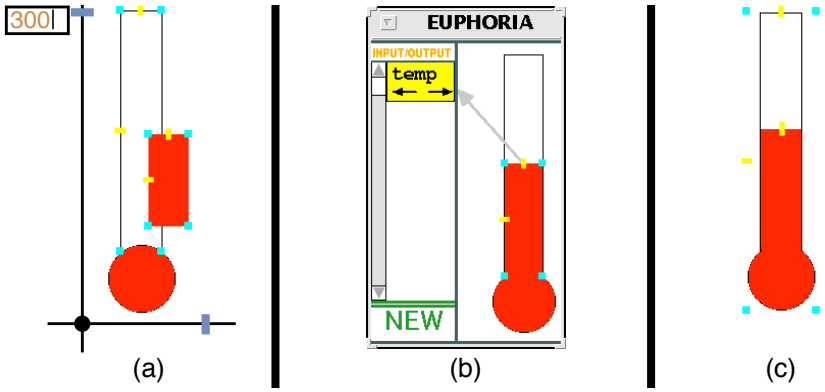

Figure 5: Creating a thermometer widget.

handle.

A widget can also have *alternatives*. For instance, a widget may have an alternative for its "selected" representation in addition to its conventional representation. Each alternative may have a set of exposed attributes. The currently displayed alternative is selected by means of an "alternative ID" widget attribute. Section 5.2 describes an example application which uses a widget with alternatives.

5 EXAMPLE APPLICATIONS

This section describes two example applications using EUPHORIA. Descriptions of additional applications can also be found in Section 7 and other publications [7][22].

5.1 Process control simulation

Maple syrup is produced by pouring maple sap into a vat and boiling away excess water until a suitable concentration level is reached (see Figure 2). A factory producing maple syrup must adjust a number of actuators controlling properties such as the incoming sap flow rate and the burner status (i.e., on or off). The process control application automates maple syrup production, controlling the factory actuators in response to sensor values. A GUI is created in EUPHORIA that displays an interactive animation of the production state to an operator who can override the system's decision through direct manipulation.

The process control application consists of three communicating modules (see Figure 1). The SENSOR module monitors conditions of the syrup production: concentration, volume, and temperature. These values are communicated to the other modules. The CONTROL module controls the actuator settings based on the sensor values, communicating the settings to the other modules. The EUPHORIA module takes as input the sensor and actuator values, animating the production display over time. Sensor and actuator information is exposed among the modules through the use of Playground published variables. End-users configure the communication among these modules at run-time by drawing logical connections between the published variables.

The display is drawn in EUPHORIA (see Figure 2), utilizing end-user defined widgets for each display component (e.g., see creation of the thermometer widget in Figure 5). Each widget has a published handle which is connected to EUPHORIA's data boundary through the use of constraints. For example, the thermometer widget has a published handle representing its temperature. Widgets are drawn using their own coordinate system, so that external modules can specify display values in terms of real world values rather than raw pixels. For example, the temperature is specified in terms of degrees Fahrenheit rather than the pixel height of the thermometer's "mercury" rectangle. This allows the SENSOR and CONTROL modules to be created completely independent of their user interface.

Note that the actuator connections between the CONTROL and EUPHORIA modules are bidirectional, allowing user interaction in the display to override decisions of the CONTROL module, in which case the CONTROL module would report the user-specified values to the SENSOR module. For example, the factory operator can adjust the valve of the incoming sap by dragging the width of the sap flow rectangle.

5.2 Medical image processing & remote collaboration

A nuclear medicine radioactive blood pool study is used to create movies of the human heart for diagnostic purposes. Each movie consists of a series of pixmap images. One problem is that ambient radiation introduces noise, making it difficult to read the images. A solution to this problem is to reduce the noise by digitally filtering the images.

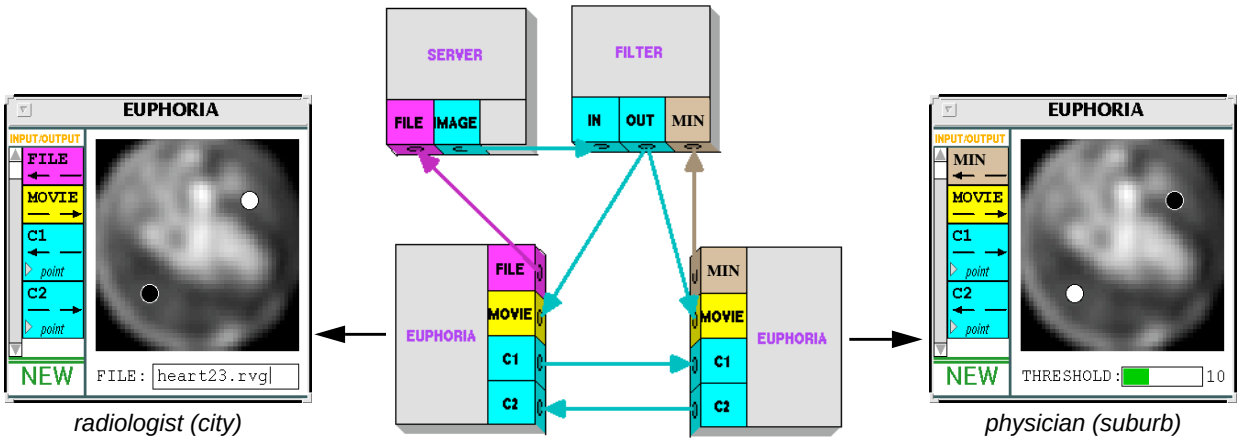

Figure 6: Medical image processing application.

An interactive filtering application is created through the use of four communicating modules (Figure 6). A SERVER module outputs a series of images of the human heart in response to a supplied file name. A FILTER module takes as input an image and outputs a filtered image based on a supplied threshold minimum. Customized, multi-user GUIs for the filtering operation are created in EUPHORIA, and are used to both control and view the filtering operation.

One possible use of this application is remote collaboration between a radiologist and an attending physician. The radiologist, located in a city hospital, specifies the file name of the blood pool study within EUPHORIA by editing a text object. The "string" attribute of the text object is published as a variable called FILE, which is connected to the FILE published variable of the SERVER module. Editing the text object causes the file name to be communicated to the SERVER module. The attending physician, located in the city's suburb, may specify the filtering threshold by either dragging a slider or editing its text value. The "width" attribute of the slider body is constrained to be equal to the "string" attribute of the slider text value through the use of a conversion constraint. The threshold value is published as a variable called MIN, which is connected to the FILTER module's MIN published variable. Based on the threshold, the FILTER module outputs a series of filtered movie images to both GUIs. Each GUI stores the images in a movie widget, with each image stored in a separate alternative of the widget.

The GUIs also have a pair of shared cursors for the physician and the radiologist to discuss areas of interest within the movie. Each person has their own cursor which they can control, plus an additional cursor showing the movement of the other person's cursor. The cursors are created by drawing circles, publishing the positions of the circles, and making the appropriate logical connections among the EUPHORIA modules. When a person moves their cursor, the published position of the cursor is communicated to the other person's GUI, moving its corresponding cursor. Additionally, a "region of interest" rectangle could be drawn and shared by the two GUIs.

Changing the displayed image of the movie (i.e., the current "frame") is achieved by making a constraint to the "alternative ID" attribute of the movie widget (not shown). The displayed frame ID can be published and shared between the two GUIs[3]. The playback rate of the movie (i.e., number of frames per second) can be specified in a number of ways. A textual frame number can be created and attached to the alternative ID, displaying the current frame number and allowing a single still frame to be specified. A slider similar to the threshold slider can be drawn and attached to the alternative ID, allowing direct manipulation of the displayed frame, range, and rate of playback. A third approach is to create a separate Playground module which increments a counter at regular intervals and connect the counter value to the alternative ID of the GUI's movies. The frame number range and rate of playback may also be specified graphically within a GUI module.

6 DESIGN SUMMARY

EUPHORIA consists of a number of software components that were designed to support the development of distributed multimedia applications. This section describes some of these components, their benefits, and how they interact. Note that this is a description of the internal design of EUPHORIA and *not* a description of the software utilized directly by end-users.

6.1 Graphics package foundation

One of the goals of The Programmers' Playground is to provide a uniform communication mechanism between computers with different hardware platforms and operating systems. For this reason, a graphics package was developed to facilitate portability of the user interfaces to different window systems/graphics libraries. The graphics package consists of a system-dependent *tiny graphics package* component and a higher level component written in terms of the tiny graphics package. The idea is to minimize and encapsulate system-dependent operations so that only a limited amount of code needs to be changed when porting to a different window system[4].

Graphics objects are defined declaratively, rather than procedurally, by instantiating C++ classes. This allows the user of the graphics package to create graphics objects of a view and let the system take care of the updates to the view. Changes to graphics objects are recorded by the system through the use of *invalidation regions*. That is, when an object changes, its old and new spatial dimensions are added to an invalidation region. Periodically, the invalid portions of graphics windows are redrawn. Double buffering is utilized for smooth animation. Redrawings of the graphics window are synchronized to minimize the amount of drawing needed to be done by an application. This allows graphics objects and images to be directly manipulated in real-time without the use of "xor patterns" that most commercial graphics editors and window systems utilize.

The event system of the graphics package was custom made to allow high level graphics processing. Processing of events and updates are interleaved, providing natural "modeless" operations. For example, in most commercial applications, selection of a menu causes all other activity on the screen to become suspended. In our graphics package, all direct manipulation can be done at the same time as animation and updates from external applications.

6.2 Communication with other Playground modules

Like all Playground modules, EUPHORIA communicates with external modules through the use of published variables. Published variables are connected to the attributes of graphics objects through the use of constraints. Over time, external input changes to the published variables are collected. At regular

[3]Due to communication delays over the Internet, the two movie displays may not be precisely synchronized.

[4]Currently, the tiny graphics package is written using only a small subset of the X window system's Xlib library [26]. It consists of approximately 1000 lines of code.

intervals of time, the set of changes to the published variables is propagated *simultaneously* into the constraint graph of EUPHORIA. Output changes to published variables are also produced within an *atomic step* operation [8] at regular intervals. These synchronizations and simultaneous updates are necessary to keep values consistent in the presence of asynchronously running modules and interprocess communication delays. Also, this synchronization greatly reduces both the amount of interprocess communication generated by the EUPHORIA module and the amount of redrawing of graphics objects in EUPHORIA.

6.3 UltraBlue constraint solver

Constraints are the heart of internal communication and other types of interaction in EUPHORIA (see Section 6.4). We developed a constraint algorithm called *UltraBlue* [20] to serve as the constraint engine of EUPHORIA. UltraBlue inherits ideas from a well known constraint algorithm called DeltaBlue [4]. Like DeltaBlue, UltraBlue is a graph-based constraint algorithm for maintaining and solving a set of constraint relationships. Constraints are *multi-way*, meaning that the computation direction of a constraint graph can be changed dynamically based on the *incremental* addition or deletion of constraints. Constraints are also *hierarchical*, meaning that different constraints have different levels of preference. Constraint preferences are used to resolve conflicts in the presence of conflicting constraint relationships.

DeltaBlue, however, does not handle cyclic dependencies of a series of constraints in a manner that is consistent with the hierarchical preference structure. In the early stages of development of EUPHORIA, it became clear that most interesting constraint relationships involve some type of cyclic dependency. UltraBlue handles cycles by changing the directionality of the constraint graph when cycles are detected. This cycle avoidance strategy considers the preference levels of constraints to maximize the number of preferred constraints enforced over constraints with lower preference levels.

Constraints in DeltaBlue are generally equality relationships among a set of values. UltraBlue provides an additional mechanism called *verification* which allows inequalities and other kinds of invariants to be maintained.

6.4 Internal constraint communication

Graphics objects in EUPHORIA have an underlying constraint representation that is essentially a directed acyclic graph (Figure 7). This graph structure allows us to have a consistent internal communication structure within EUPHORIA. That is, constraints are not only used for user-defined relationships between graphics objects, but are used for direct manipulation and internal communication among objects as well.

Figure 7a shows the internal constraint graph representation for a rectangle. A rectangle consists of a number of attributes such as “left”, “right”, and “width” represented as *constraint variables*[5] of the constraint graph. Constraints are formed among constraint variables, defining relationships to be maintained among the attributes of the rectangle. For example, the “right” attribute of the rectangle is the sum of the “left” and the “width” attributes. Whenever one or more constraint variables change, the changes are propagated through the graph, updating all connected constraint variables according to the established constraint relationships.

Other types of constraints include *stay* constraints and *active value* constraints [12]. Stay constraints serve to keep a constraint variable constant, in the absence of other constraints. These constraints are usually specified with a low preference level. When other constraints with higher preference levels are specified, the stay constraints are overridden, becoming unenforced. In Figure 7a, both “width” and “height” have associated stay constraints. This is useful, for example, in keeping the size of the rectangle constant when it is being moved by the user. Also, the “width” and “height” variables have associated

[5]Constraint variables have no direct relationship to Playground variables.

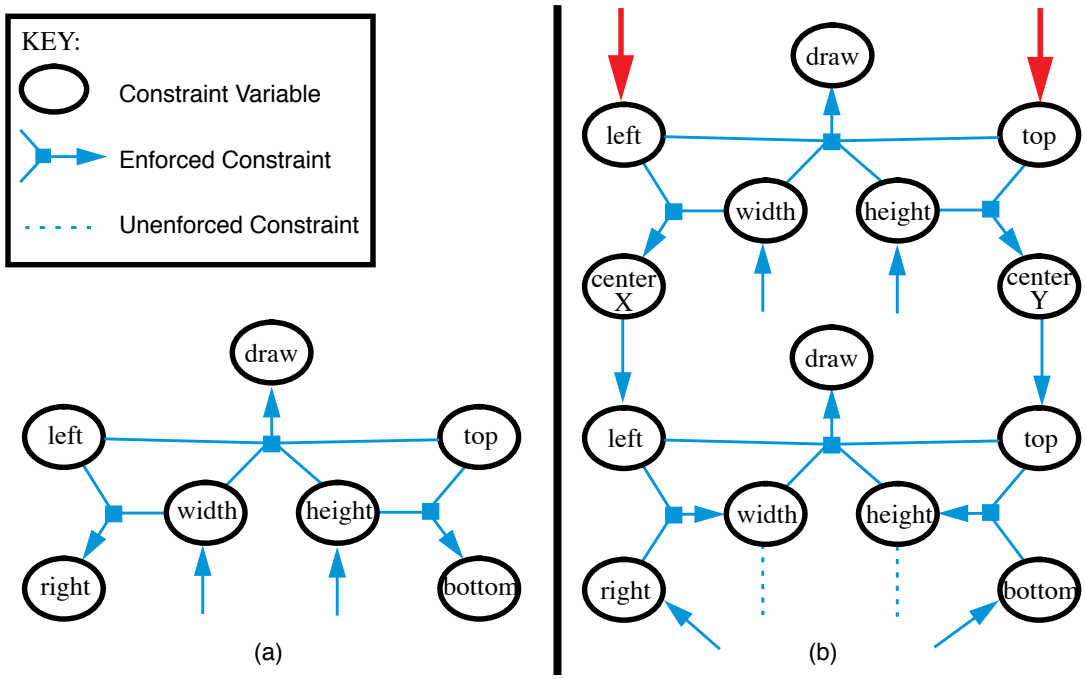

Figure 7: A) Rectangle constraint graph. B) Attaching a handle constraint graph (handle is dragged by the user).

verification methods ensuring that their values always remain non-negative. An active value constraint is used to automate the drawing process of the rectangle. Whenever the active value constraint is evaluated (as a result of connected constraint variable changes), areas of the window are invalidated for later redrawing. In Figure 7, active value constraints are shown as the constraints which compute the "draw" constraint variables.

Graphics object handles can be connected directly to the constraint graph of the graphics object under their control. In this way, one can manipulate an object using the same mechanism that is used for other types of internal communication. Figure 7b shows the constraint graph representation of a rectangle with an attached handle for its left-top corner which is being dragged by the user (i.e., resizing the rectangle). Attaching a handle simply involves creating a constraint graph for the handle and forming constraints between the handle's graph and the rectangle's graph. In this way, whenever the rectangle or the handle change position or size in any way (e.g., direct manipulation, external interprocess communication, etc.) the other is changed and redrawn to be consistent with that change. The directionality of the constraints is changed dynamically based on the addition or deletion of other constraints.

To resize the rectangle, a few other constraints are added to the graph. In Figure 7b, the "right" and "bottom" constraint variables are anchored with constant constraints. These constraints are specified with a preference level that is higher than that of the stay constraints on the "width" and "height" constraint variables. This causes the rectangle graph to be redirected, changing how the rectangle attributes are computed. In this case, the change specifies that the right-bottom corner of the rectangle should remain constant while the width and height are changed by means of the handle. The "left" and "top" constraint variables of the handle (shown at the top of Figure 7b) have attached *edit* constraints. Whenever the handle is moved by the user, the x and y coordinates of the handle are copied into these constraint variables. Propagating these values through the constraint graph results in: (1) redrawing the handle, (2) computing a new size for the rectangle, (3) redrawing the rectangle, and (4) propagation of values to any other objects that may be attached to the graph.

6.5 Communication structure overview

Figure 8 provides an overview of how both interprocess and internal communication occurs within EUPHORIA. A Playground application consists of a number of modules, including EUPHORIA, each of which have a set of published variables. Interprocess communication between modules is defined through logical connections between published variables. Whenever a published variable is changed by its module, the change is communicated to the connected variables according to the logical connections.

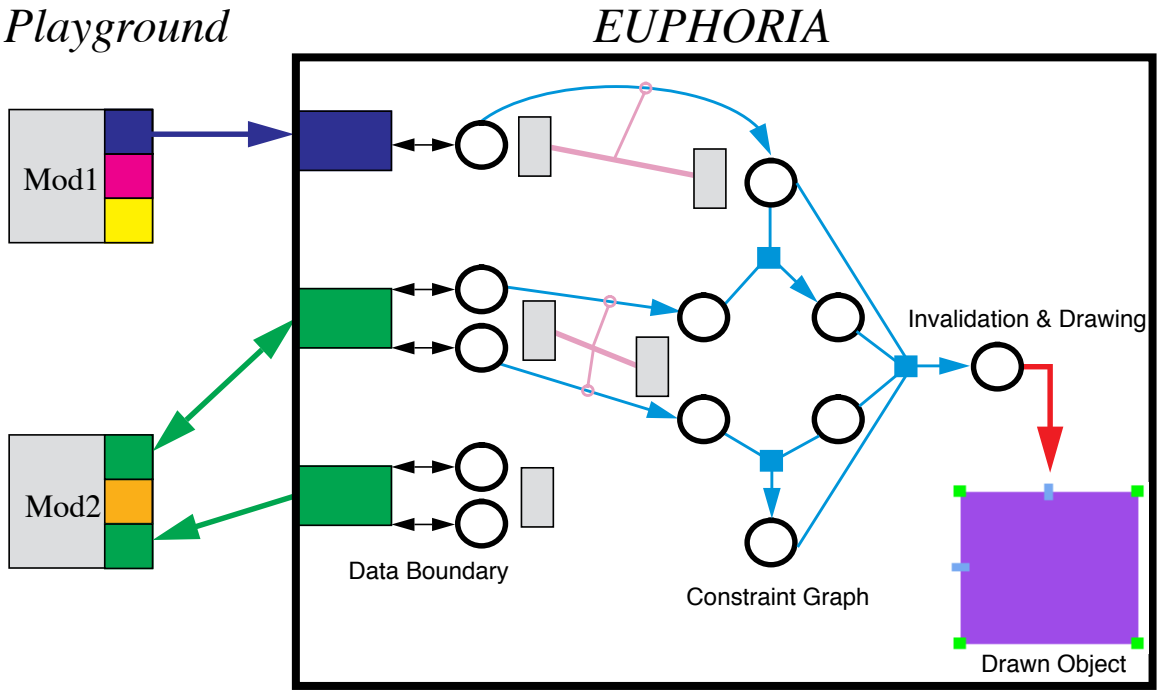

Figure 8: Communication structure of EUPHORIA.

When external values are communicated to EUPHORIA's published variables, EUPHORIA reacts by copying the values into constraint variables associated with the data boundary. These constraint variables are connected to the constraint graphs of graphics objects within EUPHORIA. The values of these variables are propagated through the connected constraint graphs, having the effect of changing graphics object attributes according to the established constraint relationships and redrawing graphics objects. In the same way, internal changes to graphics objects (e.g., direct manipulation) are propagated through the constraint graph and copied into EUPHORIA's published variables, sending the values out to external applications.

Graphics objects and EUPHORIA published variables also have associated *ports*, shown as tall rectangles in Figure 8, that are used to manage *bundles* of constraints. Ports and bundles are used for internal bookkeeping and type checking. For example, a rectangle has an upper left corner attribute that is of type "point." This attribute is actually represented as a pair of constraint variables for the x and y coordinates of the point and is managed by a port. This port cannot be connected to the port representing the width of a rectangle, since width is of type "real number." A bundle between "point" ports is actually a pair of constraints between the x and y values of the ports. Ports and bundles are also used to visualize constraints (see Section 4.4).

6.6 Run-time construction

EUPHORIA does not have separate modes for creating and running GUIs. Instead, GUIs can be created interactively at run-time and can be modified while they are being run. This gives the designer of a

GUI a sense of *instant gratification*, since one can immediately see the results of changes during construction. This also gives end-users the ability to customize a GUI (possibly created by someone else) even while its application is running! The drawing command palette and data boundary of the EUPHORIA window (see Figure 2) can be hidden, giving the end-user an unobstructed view of the GUI.

A special tool is not required to create widgets. Instead, a widget can simply be drawn in EUPHORIA and saved. One can load a EUPHORIA file, interpreting it as a widget. In this way, users do not need to learn about a different interface for creating widgets and the implementation does not need to support two different types of graphics editors.

7 FUTURE WORK

This section outlines specific plans for further enhancements to EUPHORIA in order to broaden the supported class of applications.

7.1 End-user specification of arbitrary constraint relationships

A calculator object would allow users to specify arbitrary constraint relationships among graphics objects. Like other graphics objects, a calculator has a number of variable ports, from which the user can make constraints. The ports are given names, and an editing area allows the user to specify an algebraic formula using the port variables. A multi-way constraint graph is constructed from the formula, providing a means to compute any of the variables dynamically in terms of the others. Once the user is satisfied with the configuration, the calculator can be made imaginary.

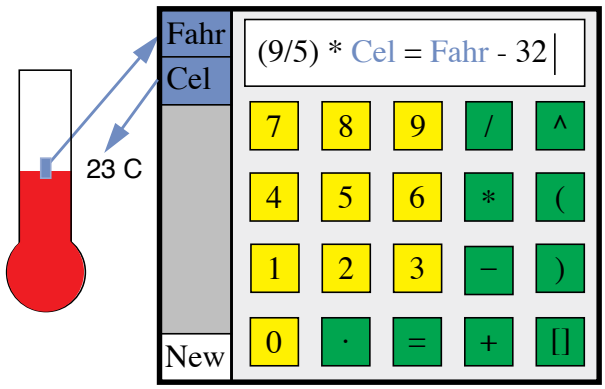

Figure 9: A calculator object for converting between temperature scales.

One use of a calculator object is a conversion between scales of measurement, such as Celsius and Fahrenheit temperatures (see Figure 9). The calculator maintains the mathematical relationship between its two ports, computing degrees Celsius when the thermometer is manipulated or computing degrees Fahrenheit when a new Celsius value is entered. Another use is the computation RGB color values of an object based on a single input value. For example, the syrup factory example described in Figure 5.1 has a widget displaying the concentration of maple syrup using a bar graph (see Figure 2). A more natural way to display this information is to have the color of the syrup in the vat appear darker as the syrup becomes more concentrated.

Calculators encapsulated inside widgets allow frequently used equations to be easily duplicated. Figure 10 shows an addition widget, built using a calculator object. By constraining handles to the three ports, one can constrain a value to be the sum of two other values. Apart from the definition of user interface modules and the encapsulation of modules to form larger units, we have not addressed the problem of end-user construction of the computational components. A visual programming language supporting the construction of Playground modules would offer the user increased flexibility in the construction of

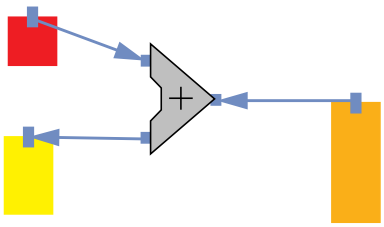

Figure 10: An addition widget.

custom applications. A promising approach to this problem would be the integration of Playground with a general purpose visual computation language based on dataflow concepts, such as the "Show and Tell" system [16].

7.2 Aggregate mappings

An aggregate is a collection of homogeneous data structures. Playground supports a number of aggregate data types including arrays and groupings. We plan to implement *aggregate mappings* in EUPHORIA, allowing end-users to create interactive visualizations of an aggregate through the use of declarative mapping rules. Visualization of an aggregate will be achieved through "mapping" the elements of the aggregate to a space. This would be accomplished by first creating a *prototype instance* of the graphical representation of an aggregate element. To establish the mapping, one will connect the attributes of a *representative element* of the aggregate to attributes of the prototype instance. The result will be a dynamically changing, interactive visualization of the aggregate's elements.

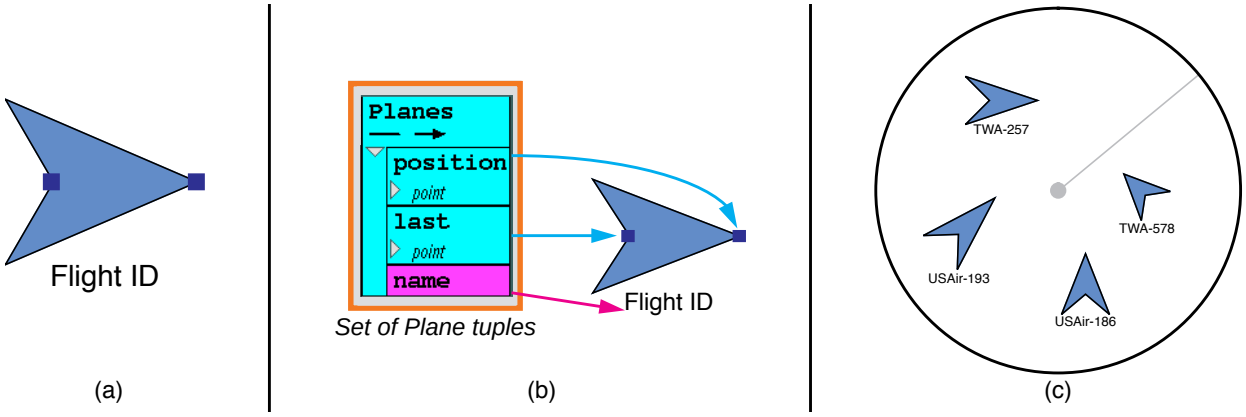

Figure 11: Creation of an aggregate mapping for an air traffic control display.

For example, an air traffic control GUI could be created in which an air traffic controller sees a circular area surrounding a centered "airport." The area contains a number of airplanes that are currently approaching or leaving the airport. An airplane is represented using a wedge shape labeled with the flight ID. The length of the wedge is used to represent the relative speed of the airplane (i.e., the longer the wedge, the faster the airplane is moving). Over time, the position and length of the airplanes are updated to display the current state [1]. An aggregate mapping for the air traffic control GUI could be created as follows. The prototype instance, a widget of an airplane, is created (Figure 11a) with three published attributes: current position, last position, and flight ID. The two positions together determine the position, length, and orientation of the airplane. Mapping rules are established by making connections from the representative element of an airplane information tuple set to the airplane widget (Figure 11b). The result is an air traffic control GUI with an arbitrary number of dynamically changing, animated airplanes (Figure 11c).

Another use of aggregate mappings is to create tabular displays (e.g., a spreadsheet). A multimedia

teleconferencing application GUI that we have previously developed [7] could also be created using tables defined by aggregate mappings (e.g., each participant of a conference is represented as a picture in an interactive table). This work is part of a project on construction of distributed multimedia applications on top of an ATM testbed [3].

7.3 Module abstractions

The module visual abstraction described in Section 2 is an intuitive mechanism for configuring Playground applications. However, for large applications it can become cumbersome to deal with many interconnected modules in this way. We have a number of enhancements to the module visual abstraction planned. Modules will be configured within EUPHORIA rather than in a separate user interface (see Figure 1). This will allow us to leverage off the existing widget mechanism to support *module encapsulation* and end-user customization of a module's appearance and interaction.

For example, the filter module described in Section 5.2 could be implemented as a number of communicating modules in a distributed image processing pipeline [7]. This would speed up computation, since frames of a movie could be processed on two or more computers in parallel. A series of several pipeline modules would make Figure 6 quite difficult to comprehend. One would be able to place the modules of the pipeline into a widget representing the overall filtering operation, publish the external variables of the widget, and make the internal pipeline modules imaginary (i.e., hide the details of the computation). The filter module would not need to appear as a "black box" as in the current implementation. Alternatively, one could customize the visual appearance of a module by drawing shapes, interactive handles, text labels, etc. within the filter module widget.

8 CONCLUSION

We have described EUPHORIA, a user interface management system that supports end-user construction of direct manipulation, distributed multimedia applications. No programming is required to create end-user GUIs or to configure the communication.

A version of the Playground system exists for creating distributed software modules in the C++ language on the Solaris and Silicon Graphics IRIX operating systems. The run-time system handles communication over sockets using TCP/IP. Supported features include dynamic end-user configuration of applications, separation of communication from computation, separation of active and reactive control, and migration of running modules to other processors.

The graphics package described in Section 6.1 has been implemented in X windows. Using this graphics package, we have implemented a "connection manager" direct manipulation graphical interface for managing communication configuration. We have also implemented EUPHORIA, using the graphics package and the UltraBlue constraint solver (Section 6.3). Supported features include real-time direct manipulation graphics, constraint-based editing and visualization, imaginary alignment objects, user-definable types, and user-definable widgets with alternative representations.

In Spring 1995, The Programmers' Playground and EUPHORIA were used to teach an undergraduate course, CS333, at Washington University on the topic of distributed programming environments. The Process Control Simulation and Medical Image Processing application described previously were among the assignments from the course.

The Playground World Wide Web site [6] contains general information, live interactive demonstrations of both the Programmers' Playground and EUPHORIA, and course materials from CS333.

ACKNOWLEDGMENTS

We thank EUPHORIA users, including the students in CS333 for their useful comments. We thank

Bala Swaminathan and Ram Sethuraman for their work in developing the Playground library. This research was supported by National Science Foundation grants CCR-91-10029 and CCR-94-12711.

REFERENCES

[1] Amir Aboueinaga. TRW Sr. Staff Engineer and FAA Consultant. Personal Communication.

[2] A. Borning. Thinglab - A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353-387, October 1981.

[3] Jerome R. Cox, Jr., Mike Gaddis, and Jonathan S. Turner. Project Zeus: Design of a Broadband Network and its Application on a University Campus. *IEEE Network*, pages 20-30, March 1993.

[4] B. Freeman-Benson, J. Maloney, Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54-63, 1990.

[5] David Gelernter and Nicholas Carriero. Coordinations Languages and their Significance. *Communications of the ACM*, 35(2):97-107, February 1992.

[6] Kenneth J. Goldman, et. al. "Welcome to the Programmers' Playground!" http://www.cs.wustl.edu/cs/playground/.

[7] Kenneth J. Goldman, T. Paul McCartney, Ram Sethuraman, and Bala Swaminathan. The Programmers' Playground: A Demonstration. In *Proceedings of the 1995 ACM International Conference on Multimedia*, November 1995. To appear. See also the conference CD-ROM proceedings for a longer version.

[8] Kenneth J. Goldman, T. Paul McCartney, Ram Sethuraman, Bala Swaminathan, and Todd Rodgers. Building Interactive Distributed Applications in C++ with The Programmers' Playground. Washington University Department of Computer Science technical report WUCS-95-20, July 1995.

[9] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ram Sethuraman. The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications. *IEEE Transactions on Software Engineering*. To appear.

[10] Michael M. Gorlick and Rami R. Razouk. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering*, May 1991.

[11] Brent Hailpern and Gail E. Kaiser. Dynamic Reconfiguration in an Object-Based Programming Language with Distributed Shared Data. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 73-80, May 1991.

[12] Tyson R. Henry and Scott E. Hudson. Using Active Data in a UIMS. In *Proceedings of the ACM Symposium on User Interface Software*, pages 167-178, October 1988.

[13] Ralph D. Hill. Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *ACM Conference on Human Factors in Computing System*s, pages 335-342, May 1992.

[14] Dan Ingalls, Scott Wallace, et. al. Fabrik: A Visual Programming Environment. In *OOPSLA Conference Proceedings*, pages 176-190, September 1988.

[15] J. Jagadeesh and Y. Wang. LabVIEW. Product Review, *Computer*, February 1993.

[16] T. D. Kimura, J. W. Choi, and J. M. Mack. A Visual Language for Keyboardless Programming. Washington University Department of Computer Science technical report WUCS-86-6, June 1986.

[17] Jeff Kramer, Jeff Magee, and Keng Ng. Graphical Configuration Programming. *IEEE Computer*, 22(10):53-65, October 1989.

[18] Jeff Kramer, Jeff Magee, and Morris Sloman. Configuring Distributed Systems. In *Proceedings of the 5th ACM SIGOPS European Workshop*, September 1992.

[19] Frank Ludolph, Yu-Ying Chow, et. al. The Fabrik Programming Environment. In *Proceedings of the IEEE*

Workshop on Visual Languages, pages 222-230, 1988.

[20] T. Paul McCartney. User Interface Applications of a Multi-way Constraint Solver. Washington University Department of Computer Science technical report WUCS-95-22, August 1995.

[21] T. Paul McCartney and Kenneth J. Goldman. EUPHORIA Reference Manual. Washington University Department of Computer Science technical report WUCS-95-19, July 1995.

[22] T. Paul McCartney and Kenneth J. Goldman. Visual Specification of Interprocess and Intraprocess Communication. In *Proceedings of the 10th International Symposium on Visual Languages*, October 1994, pp. 80-87.

[23] B. A. Myers, et al. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, 23(11):71-85, November 1990.

[24] J. K. Ousterhout. TCL: An Embedded Command Language. Computer Science Division (EECS), University of California, Berkeley, CA, January, 1990.

[25] J. M. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151-174, 1994.

[26] Robert W. Scheifler and Jim Gettys. The X Window System. Technical Report MIT/LCS/TR-368, MIT Laboratory for Computer Science, October 1986.

[27] David L. Tennenhouse, et. al. A Software-Oriented Approach to the Design of Media Processing Environments. In *Proceedings of the International Conference on Multimedia Computing Systems*, pp. 435-444, Boston MA, May 1994.