Visual Specification of Interprocess and Intraprocess Communication

T. Paul McCartney and Kenneth J. Goldman *
Department of Computer Science
Washington University
St. Louis, Missouri 63130

Abstract

We present a visual specification language for constructing distributed applications and their direct manipulation graphical user interfaces. Each distributed application consists of a collection of independent modules and a configuration of logical connections that define communication among the data interfaces of the modules. Our specification language uses a single visual mechanism that allows end-users to define interprocess communication among distributed modules and to define intraprocess communication among objects within a module. This visual language provides a general encapsulation/abstraction mechanism and is designed to support dynamic change to the communication structure. User interfaces are completely decoupled from the module(s) they control.

1 Introduction

Distributed multimedia applications supported by a global electronic infrastructure have tremendous potential for providing users with customized communication and computation environments. These applications include remote collaboration, information and resource sharing, and access to broadcast media (Figure 1). The 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 system programmers will anticipate the needs of all users. Therefore, empowering end-users to create their own customized communication and computation environments is an important challenge. Any successful approach to this problem will require a visual language that integrates distributed application configuration and user interface construction. This will involve the integration of all aspects of communication, from *intraprocess communication* (e.g., creating constraints among graphical objects in a display) to *interprocess communication* (e.g., establishing teleconferencing connections). This paper presents a visual specification language for establishing both intraprocess and interprocess communication using a uniform "connection" abstraction. Intraprocess communication is achieved by connecting attributes of simple and compound graphical objects using constraints and grouping abstractions. Interprocess communication (among diverse distributed modules) is accomplished by managing logical connections among independent modules.

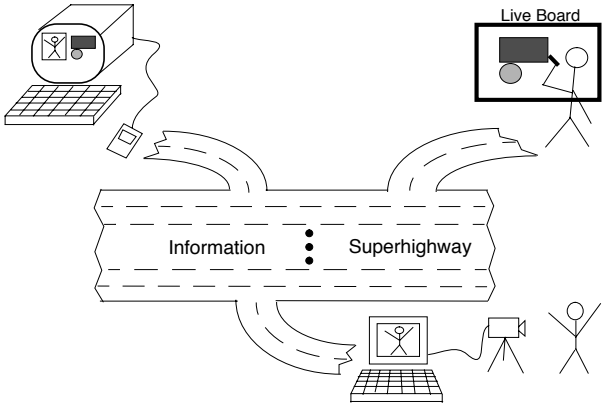

Figure 1: Distributed multimedia applications

A single visual mechanism with a consistent semantics is used for all aspects of communication. The user thinks at a high level about "plugging together" the components of a distributed multimedia application, and is not concerned with the low level implementation details of such a system. The system is free to make choices about how to implement the communication specified by each connection in the configuration.

*This research was supported in part by National Science Foundation grants CCR-91-10029 and CCR-94-12711. E-mail: paul@cs.wustl.edu and kjg@cs.wustl.edu

1.1 I/O abstraction

Our model of interprocess communication is called *I/O abstraction* [6]. Each module in a system has a module *boundary* [1] containing values (published data structures) that may be externally observed and/or manipulated. A distributed application consists of a collection of independent modules and a configuration of logical connections among the published values at module boundaries. Whenever a module updates one of its own published data items, communication occurs implicitly according to the logical connections.

I/O abstraction communication is declarative, rather than imperative. One declares direct high-level logical connections among the data items of individual modules, as opposed to directing communication within the control flow of the module. This makes implicit communication possible. Output is essentially a byproduct of computation, and 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 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 allows the run-time system to handle communication more effectively.

1.2 Objectives

We claim that the separation of computation from communication achieved by I/O abstraction can form the basis of tools that allow end-users to create sophisticated customized distributed applications from computational building blocks. To support this claim, this paper presents a visual language for specifying the communication structure of a distributed application. Our visual language addresses all aspects of communication, including interprocess communication among independent distributed modules, module boundary declarations, and intraprocess communication among objects within a single module. Our treatment of interprocess communication is completely general, while our treatment of intraprocess communication concentrates on the specification of sophisticated direct-manipulation graphical user interfaces that interact with multiple independent modules in a distributed application.

[1] In other papers [6, 18] 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 interfaces for visual languages, we use the term *boundary* in order to avoid confusion with user "interfaces" and a visual "presentation."

In the intraprocess communication area, we have concentrated on the user interface construction problem because it is critical for supporting the kinds of distributed multimedia application we envision. However, our connection oriented visual communication language would also blend nicely with general purpose visual computation languages based on dataflow concepts, such as the "Show and Tell" system [11].

Our visual language is designed to support end-user configuration of distributed multimedia applications on top of *The Programmers' Playground,* a software library and run-time system we are developing to support the I/O abstraction programming model. The user of our visual specification language does not need to write any source code to establish communication or know the details of how the communication works.

The specification of a GUI is created as an independent module using a graphics editor. At run time, the user establishes logical connections among the GUI module boundary and the boundaries of other modules in the system in order to configure a complete customized distributed application.

The development work is being conducted in the context of an ATM network being deployed on the Washington University campus [3].

1.3 Overview

The remainder of this paper is organized as follows. In Section 2, we provide an overview of related work. We present our visual communication language in the context of an air traffic control example introduced in Section 3. Section 4 discusses visual specification of intraprocess communication. In our case, intraprocess communication can be established through constraints between graphics objects or through data boundaries of encapsulated widgets (created by an end-user through direct manipulation). Section 5 describes visual specification of interprocess communication between distributed modules through data boundaries. Both intraprocess and interprocess communication are specified with a *connection oriented* visual abstraction. Communication and visualization of data *aggregates* is also addressed. Section 6 discusses our current implementation status.

2 Related work

We are not aware of other visual specification languages that integrate all aspects of communication in support of end-user construction of customized distributed multimedia applications with user-specified graphical interfaces. In this section we highlight some

of the related work in the area of coordination languages for configurable distributed systems and in the area of visual specification of user interfaces.

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 [13] 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 [12]. However, the modules of the system must still be concerned with when to send or receive messages on these ports. In Polylith [15], a configuration is expressed using "module interconnection constructs" that establish procedure call bindings among modules in a distributed system. CONCERT [19] provides a uniform communication abstraction by extending several procedural programming languages to support the Hermes [17] distributed process model. PROFIT [10] provides a mixture of data sharing and RPC communication through *facets* with data and procedure *slots* that are bound to slots in other facets during compilation. Extensions to PROFIT enable dynamic binding of slots in special cases [8]. The Weaves system [7] provides a configuration mechanism based on dataflow.

The above systems adopt a given communication model and concentrate on the configuration problem. Here, we have taken a more comprehensive approach by developing the configuration mechanism and communication model (I/O abstraction) concomitantly, in order to achieve a more effective separation of communication and computation.[2]

Much work has been done in the area of user interface construction. Here, we mention three systems that are representative of this paradigm. The Thinglab system [2] uses multiway 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. The Garnet system [14] 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. The RENDEZVOUS project [9] concentrates on the separation of the user interfaces from their applications. RENDEZVOUS is a transition from purely user interface oriented systems to systems that attempt to decouple the construction of the graphical user interface from their applications.

[2]See the technical report version of [6] for a comparison of I/O abstraction with other communication models.

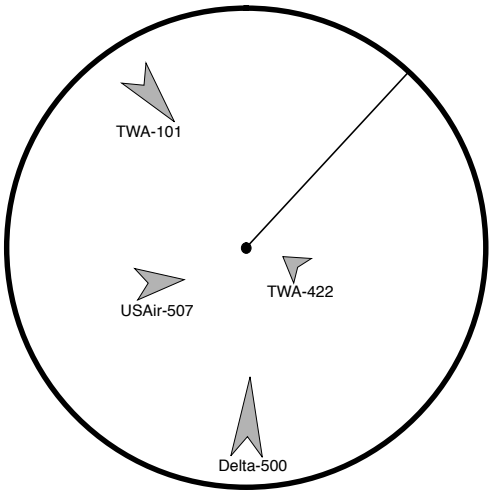

Figure 2: Air traffic control example application

3 Example application

We present our visual communication language in the context of a "toy" example of an air traffic control system. The air traffic control system consists of three communicating modules [1]: a *radar* module that gets information about the position and identity of the set of current airplanes, a *radio* module that is used to coordinate audio communication between the pilots and the air traffic controller, and a *graphical user interface* (GUI) module that displays the current state of the airplanes as shown in Figure 2.

The air traffic controller sees on the display 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 relative speed of the airplane (i.e. the longer the wedge, the faster the airplane is moving). Over time, the position

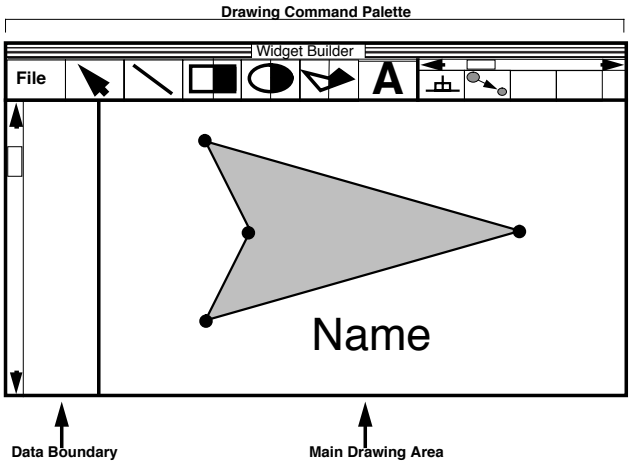

Figure 3: Graphical user interface editor

and length of the airplanes are updated to display the current state. The user of the GUI can communicate with the pilots through audio radio channels. By clicking on airplanes with the mouse, a "focus" set of flight IDs is selected. This action establishes the subset of pilots with whom the user wishes to speak.

3.1 Task outline

We illustrate our visual language in the context of a graphical editor that we are developing. Using this editor, we will describe in a bottom-up fashion how one can construct the air traffic control application. First, we define a graphical widget to represent an airplane. This involves drawing the widget, creating constraints on the shape of the widget, and establishing a data boundary encapsulation through which the widget can be manipulated.

Next, we define the air traffic control GUI. This involves drawing the GUI and establishing a data boundary through which the GUI module communicates with external distributed applications. This data boundary includes an audio channel between the pilots and the controller, a set of current airplane tuples, and a set of currently selected planes. To visualize the set of airplanes, we define an *aggregate mapping* from a set of airplane tuples to the coordinate system of the GUI. Similarly, we create an aggregate mapping from selected airplanes of the GUI to external modules.

Finally, we configure the air traffic control module with the radar and radio modules with logical connections between data values in the modules' data boundaries. This configuration, performed at run-time with a visual user interface, completes the construction of the distributed application.

The editor consists of three parts (see Figure 3). The body of the window is used for drawing widgets or graphical user interfaces. 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). Among these widgets are *imaginary* alignment objects such as a "perpendicular" object used maintain a right angle between two lines. The left side of the editor window contains the data boundary portion of editor. For widgets, the data boundary defines the set of attributes that can be used externally by the containing user interface to control the widget appearance. For graphical user interfaces, the data boundary defines the set of data structures that can be externally manipulated by external I/O abstraction modules.

4 Intraprocess communication

This section describes how users create simple and compound objects of a user interface and define relationships among those objects. Relationships include equality constraints and encapsulation of graphics object groups by means of a data boundary.

4.1 Graphics primitives and attributes

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 an object is "selected".

Attributes of a graphics object are visually represented as tags that are positioned in appropriate places relative to the graphics object (Figure 4). The user can establish relationships (e.g., constraints) among the attributes of the graphics objects by using the tags as data ports for user interaction. For novice users, tags might be labeled textually, but the text could be hidden for experienced users to avoid clutter.

4.2 Widgets

A widget is a compound graphics object that is a grouping of graphics objects with a subset of exposed attributes. One may think of a widget as a "module" of graphics shapes with a "data boundary" of externally readable/writable attributes. The values of the attributes in a widget's data boundary are the only means of controlling or viewing the state of the widget externally. Widgets are created visually by end-users. As with other graphics objects, the external attributes of a widget can be viewed, revealing tags which can be used in forming connections to the widget.

A widget can have multiple visual representations that we call *alternatives*. For instance, a widget may

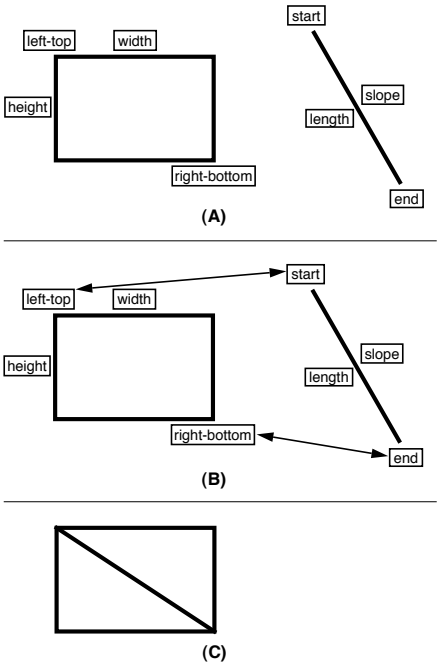

Figure 4: A) Rectangle and line with exposed attributes. B) Creating constraints between the rectangle and line. C) Result of satisfying the constraints.

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 a standard "alternative" widget attribute.

For example, an airplane in our air traffic control example can be defined as a widget. With the mouse, the user draws the outline of the polygon consisting of four points, just as in other graphics editors such as *MacDraw* or *xfig*. A textual name label for the identification of the airplane is created and positioned under the polygon (Figure 3).

4.3 Spaces

A *space* is a coordinate system that contains graphics objects. To simplify user interface construction, our visual language allows end-users to define multiple spaces with independent coordinate units, origins, and clipping regions. Aggregate values such as sets and arrays can be *mapped* onto the space or mapped from the space to an external variable in the data boundary with a visual mechanism (see Section 5.2).

4.4 Constraints

Constraints are a simple, yet powerful, way to specify relationships among graphical objects. Establishing an equality constraint between graphical objects is

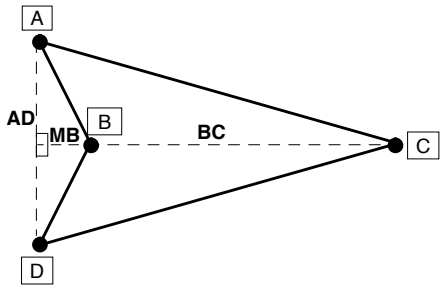

Figure 5: Imaginary object constraints

accomplished by simply making a connection between a pair of exposed attributes of two graphical objects. Figure 4b and 4c show how a user would specify a line to be constrained between the corners of the rectangle.

However, many desirable constraint relationships cannot be established by direct equality constraints. For this reason, our specification language supports the concept of "imaginary objects." Imaginary objects are invisible shapes which 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 create indirect constraints between graphics objects visually using the same mechanism used to create direct constraints between the attributes of visible objects.

To define the shape of the airplane, we create imaginary line segments $\overline{AD}$, $\overline{BC}$, and $\overline{MB}$, where M is the midpoint of $\overline{AD}$ (note: the lines are not actually labeled in the editor). Then we constrain $\overline{MB}$ and $\overline{BC}$ to be co-linear and constrain $\overline{AD}$ and $\overline{MB}$ to be perpendicular using a "perpendicular" imaginary alignment object which is predefined in the widget library (see Figure 5). All relationships are declared visually.

4.5 Intraprocess data boundaries

Once the component shapes and internal constraints of a widget are specified, it is "packaged" for later use in a graphical interface. When used, the internal details of the widget are hidden from the user. The appearance of the widget is controlled strictly through its data boundary of "exposed" attributes. In our graphics editor, the widget boundary is declared by establishing connections between attributes of the widget and the data boundary area of the editor. Each connection creates a data boundary attribute that is shown as a rectangle containing the user-specified attribute name.

In the air traffic control example, the position, orientation, length, and name of each airplane is de-

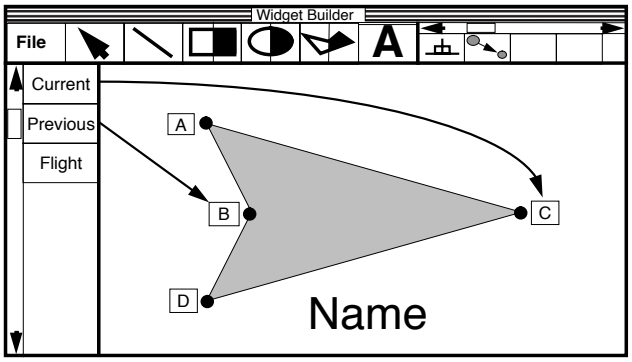

Figure 6: Publishing external widget attributes

termined by an external data source (recall that the length of an airplane is proportional to its speed). We create three widget attributes *current, previous,* and *flight* in the data boundary of the widget. *Current* represents the current position of the airplane; *previous* represents the previous position of the airplane; *flight* represents the flight ID of the airplane. To satisfy position, orientation, and length requirements, we can think of the current position of the airplane being at the "nose" of the airplane; the previous position of the airplane is at the "tail" of the airplane. Given the current position and previous position of the airplane, the orientation and length requirements are satisfied. The greater the difference between the current position and the previous position, the greater the distance between the nose and the tail of the airplane widget.

We publish attributes of the airplane widget as shown in Figure 6. When the airplane widget is instantiated in a user interface, only the *current, previous,* and *flight* attributes are externally exposed. When the exposed attributes of the widget are viewed, they are revealed in place relative to their position within the widget.

5 Interprocess communication

An I/O abstraction module is an independent process that has a data boundary consisting of a set of exposed variables. Modules communicate exclusively through the variables in their data boundaries. A Playground user does not need to understand the details of interprocess communication to create distributed software modules. A Playground module is simply a program written in a standard programming language (e.g., C++) using the Playground library.

Communication between modules is specified through *logical connections* between variables of the module boundaries. When the value of a published variable changes during the course of execution, the changed value is implicitly communicated to all connected variables in other modules. The details of how the communication is handled is 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.

5.1 Interprocess connection manager

Playground modules and logical connections have a visual representation in our visual specification language. A Playground module (that is, an active process) is represented as box with a set of data "plugs" for each variable in the module's boundary. The color of each plug represents its type. Logical connections are represented as arrows between pairs of variables in module data boundaries (see Figure 7). The metaphor is that of wiring together the components of a stereo system, where the color of each cord denotes the type of information it carries.

5.2 Aggregates

The Playground system supports *aggregates* (compound data types such as sets and arrays) in a module's data boundary. Visualization of an aggregate is accomplished in our language through "mapping" the elements of the aggregate to a space in a graphical user interface. This is accomplished by first creating a *prototype instance* of the graphical representation of an element of the aggregate within a space. To establish the mapping, one connects the attributes of a *representative element* of the aggregate to attributes of the prototype instance.

For example, in our air traffic control GUI, we would like to establish a mapping between the "planes" aggregate from the radar module to the space of the GUI. First, we select our previously defined airplane widget and denote it as the prototype instance (see Figure 8). Next, we expose the representative element of the planes variable, which consists of a tuple containing variables "position", "last", and "name". Finally, we make the appropriate connections to the airplane widget prototype.

We also establish an aggregate mapping from the space of the GUI to the focus module boundary value (recall that the "focus" value is a subset of airplane flight IDs which are selected by the user). This mapping is defined through the use of a "selected" airplane widget alternative. We connect the flight ID

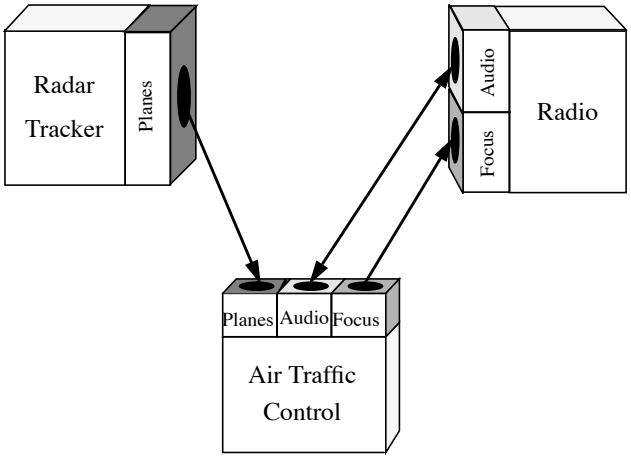

Figure 7: Air traffic control distributed configuration

attribute from the widget's selected alternative to the focus value in the module boundary.

When an airplane widget is selected by the air traffic controller, the widget changes visual representations toggling between the conventional and selected alternatives. When an airplane widget is currently displayed using the selected alternative, its flight ID is included in the focus set due to the aggregate mapping. Thus, the user can select a subset of airplanes dynamically. External distributed modules may view this value. In our example, it is used to select appropriate radio channels within the radio module.

5.3 Module configuration

The radar and radio modules, having been separately defined using the Playground library, are configured with the air traffic control module to complete the application. The data boundary of the radar tracker module consists of a single readable "planes" variable, which is a set of tuples containing current airplane status. The data boundary of the radio module consists of a read/write "audio" variable and a writable "focus" variable. Note that the audio variable is a continuous data type, but communication is specified in the same way as discrete data types.

The graphics editor from Figure 8 automatically creates a module for the air traffic control GUI without user programming. With a connection manager GUI[3], we can configure the modules of the air traffic control application dynamically at run-time (see Figure 7). We use this GUI to create logical connections between the radar, radio, and air traffic control modules, establishing interprocess communication.

[3]The connection manager GUI could be created using the graphics editor described in this paper.

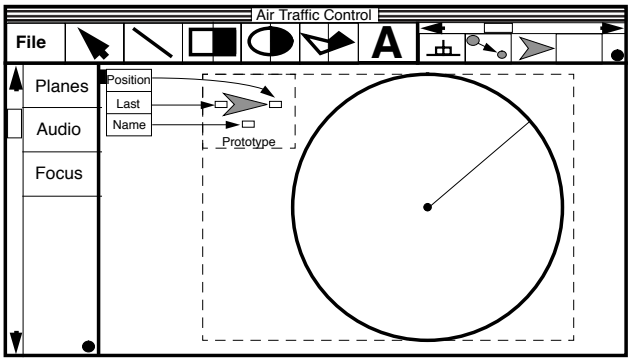

Figure 8: Mapping an aggregate to a space

5.4 Extensibility and module reuse

The air traffic control module is independent of the radar and radio modules of the application shown in Figure 7. Because of this independence, it is easy to use this module for a slightly different purpose. Suppose that in addition to the air traffic control display of the current state of the airplanes, we wish to have a projected display of the future trajectory of the airplanes. That is, we want a simulation that repeatedly extrapolates forward in time at an accelerated rate from the current state to a future state (e.g., one minute ahead). This can be accomplished by creating a "simulator" module which takes as input the current state of the airplanes and outputs the interpolated future state of the airplanes. As seen in Figure 9, the end-user can simply create a separate instantiation of the air traffic control GUI to display the simulated state in addition to the current state display (these GUIs would be displayed in separate windows). Also note that the output of the radar tracker is a *multicast connection* to two different modules in this configuration. The Playground system automatically handles the details of this communication at run-time, without any special consideration from the implementor of the radar tracker module.

6 Implementation status

Our visual specification language is a tool for constructing distributed multimedia applications. A version of the Playground system exists for creating distributed software modules in the C++ programming language. Also, we have designed and implemented a customized graphics package which will be used as the foundation of the user interface management system. This graphics package is currently implemented on top of the X window system [16], but it is not designed exclusively for X windows. Using the graphics

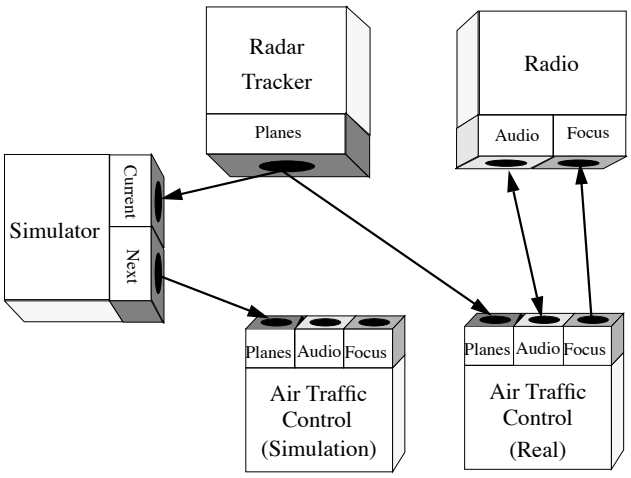

Figure 9: Simulator

package, we have implemented the "connection manager" direct manipulation graphical interface for managing interprocess communication Playground system (see Figures 7 and 9). As a foundation for our intraprocess communication, we have implemented an efficient incremental constraint solver. The solver uses ideas from the DeltaBlue algorithm [4] for local constraint strength propagation, but it resolves cycles of constraints intelligently and has improved time complexity.

7 Conclusion

We have presented a visual specification language for the configuration of distributed multimedia applications. Our visual language supports the specification of communication among components of a distributed application at all levels, from communication among graphics primitives within a user interface to communication among large modules distributed across multiple processors. The language also supports encapsulation at each level, and allows the user to expose information at the data boundary for use at the next level of abstraction. The visual mechanisms and semantics are consistent across all levels.

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 and J. Maloney Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[5] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[6] Kenneth J. Goldman, Michael D. Anderson, and Bala Swaminathan. The Programmers' Playground: I/O abstraction for heterogeneous distributed systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 363–372, January 1994. Long version available as Washington University technical report WUCS–93–29.

[7] 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.

[8] 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.

[9] Ralph D. Hill. Abstraction-link-view paradigm: using constraints to connect user interfaces to applications. In *ACM Conference on Human Factors in Computing Systems*, pages 335–342, May 1992.

[10] Gail E. Kaiser and Brent Hailpern. An object-based programming model for shared data. *ACM Transactions on Programming Languages and Systems*, 14(2):201–264, April 1992.

[11] T.D. Kimura, J.W. Choi, and J.M. Mack. A visual language for keyboardless programming. Technical Report WUCS–86–6, Washington University in St. Louis, June 1986.

[12] Jeff Kramer, Jeff Magee, and Keng Ng. Graphical configuration programming. *IEEE Computer*, 22(10):53–65, October 1989.

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

[14] B. A. Myers, et al. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[15] J.M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

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

[17] R.E. Strom, D.F. Bacon, A.P. Goldberg, A. Lowry, D.M. Yellin, and S. Yemini. *Hermes: A Language for Distributed Computing*. Prentice-Hall, 1991.

[18] Bala Swaminathan and Kenneth J. Goldman. Dynamic reconfiguration with I/O abstraction. Technical Report WUCS–93–21, Washington University in St. Louis, August 1993.

[19] Shaula A. Yemini, German S. Goldszmidt, Alexander D. Stoyenko, and Langdon W. Beeck. CONCERT: A high-level-language approach to heterogeneous distributed systems. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 162–171, 1989.