The Global Desktop: A Graphical Composition Environment for Local and Distributed Applications

Tien Huynh Charanjit Jutla Andy Lowry Robert Strom Daniel Yellin

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
Email: dmy@watson.ibm.com; Phone: (914) 784-7948

November 16, 1993

1 Introduction

We believe that two of the most profound changes that will permeate personal computing in the coming years are the advent of *collaborating software components* and *distributed access to information and services.*

Most applications today are monolithic, self-contained pieces of software. Developers often have difficulty maintaining and supporting existing functionality, let alone addressing market pressure for new features. A promising alternative is to create smaller, specialized software components from which customized applications can be built through composition. Rather than including a full-featured text editor for editing text cells, a spreadsheet program may rely on an independent software component that is specialized for that purpose. The same text editor component might also be reused with other software components to create an electronic mail application, and other compatible text editors could be substituted as a new way of customizing the spreadsheet application.

Increasingly, data and services are disseminated around the world through computer networks. Although client/server and other distributed computing technologies have grown dramatically in the last few years, the average computer user, for the most part, still uses a standalone computer system. Nonetheless, even these users are increasingly using on-line information services to gather information from outside electronic sources, and to connect to other on-line services.

Many programming environments focus on languages and tools for composing independent and possibly distributed applications, and thereby provide a technological basis for the sort of evolution we envision. Whereas some environments focus on the *static* composition of modules, others focus on the *dynamic* composition of running *processes.* Examples of the former include module interconnection languages [10] and software integration toolkits [11]. Examples of the latter include client/server paradigms which allow client applications to dynamically link to and request services from running server programs [9, 14, 8], and architectures such as Tooltalk [15], HpSoftBench [4], and Field [12]. These latter systems allow programs to broadcast events to other applications, and to listen to events being broadcast from other applications.

The Global Desktop (GD) described in this paper takes the dynamic composition approach one step further: it brings the power of application composition to non-programmer end-users. The applications being linked can reside anywhere on the network, and the end-user need not be programming literate. Indeed, the end-user uses simple graphical gestures to indicate the intended composition and the Global Desktop infrastructure does the rest. The user of the Global Desktop can:

- customize her environment by linking independently created software components together, creating synergistic solutions for specific tasks;
- distribute access to information and services on her machine to other users, and obtain access to information and services provided by other users;

Joe,
Here is access to the car auction I told you about. Plug the connector into your bidder. Have fun! Donna

Figure 1: Electronic mail from Donna to Joe carrying a "wire plug" representing a collaboration capability.

- search (in both local and remote repositories) for components that perform specific functionality.

Most programming language expertise has focused on the right abstractions to present to expert programmers for creating modular and distributed programs. The Global Desktop addresses the question: how can we reflect the notions of modularity and connectivity in a natural and intuitive way to non-programmer end-users? This will be addressed in section 2, where we describe the end-user model in detail.

Creating such an end-user paradigm creates many system challenges as well. The designers of the Global Desktop have attempted not only to support a natural and intuitive end-user paradigm, but also to make it easy for programmers to incorporate their applications into the Global Desktop, and for applications written to disparate interaction paradigms to collaborate. This is accomplished by means of a flexible architecture, a separation of *contractual* interfaces from application interfaces, and tools that automatically generate code that sits between applications and the desktop infrastructure. The implementation of the Global Desktop is described in section 3. In the remainder of this section, we will present an example of how the end-user interacts with the Global Desktop (GD), utilizing a scenario which is fully implemented as described, in our prototype.

Consider an end-user, Joe, who wants to buy a car. Initially we find Joe using two applications on his desktop: a classified ads browser called "CLASSY," and an auto-reports browser called "AUTO-GURU" that provides comparison-based information on automobiles. These applications both allow the user to choose an entry (e.g., a particular ad or a particular vehicle description) to view. Attached to each application window is a *collaboration panel*, which shows plug and socket *connectors*. These connectors indicate that the applications can be linked to other applications to perform collaborative tasks. In particular, both CLASSY and AUTO-GURU support the "synchronize views" collaboration.

After Joe repeatedly manually updates AUTO-GURU to view information about the car he is currently viewing in CLASSY, he decides to start letting the applications do the updating for him, by "wiring" the two "synchronize views" connectors together. From then on (until he detaches the wire and terminates the collaboration), whenever Joe selects an ad in CLASSY, AUTO-GURU immediately displays its corresponding entry as well.

Soon, Joe becomes overwhelmed by the number of classified ads that cost more than he can afford. Although CLASSY does not supply any built-in controls to limit the selection, it does offer a "filter" collaboration via a socket connector in its collaboration panel. Joe creates a wire and plugs it into this socket, then drops the other end of the wire onto an icon labeled "search" in a "REPOSITORY" window that he always keeps on his desktop. The repository scans the network and locates a collection of applications that provide a variety of filtering capabilities. Joe selects one that allows him to set up arbitrary numeric tests, attaches it the wire coming from CLASSY's "filter" socket, and types "Price < \$1000" into the filter. The list of ads displayed by CLASSY is immediately pruned substantially. However, there are still many ads for cars that are too distant for Joe to be interested. Joe pulls out another filter from the search list created by his repository, and cascades this filter with the other one. The new filter shows a map and allows Joe to "lasso" a region of the map with his mouse to limit CLASSY to ads for cars within that region.

Suddenly, Joe notices that he has mail from his good friend Donna. Figure 1 shows the letter Joe retrieves from his electronic mailbox.

Donna just sent Joe the capability of interacting with an auction; someone on the network is selling cars, and has provided the capability to bid on the electronic auction by distributing "connectors" to his electronic auction application. Donna has forwarded such a connector to Joe. When Joe removes this connector icon from the letter and and plugs it in to a bidding application that supports the "auction collaboration" (which he obtains by searching his repository again), Joe will be able to bid for the cars. Joe, with a few graphical gestures, has set up a distributed collaborative session!

2 The end-user interaction paradigm

2.1 Desktop entities

There are three sorts of entities in the GD: *components*, *data*, and *wire connectors*. This section will elaborate on these entities, as well as the fundamental notion of desktop *collaboration*.

2.1.1 Components and Collaboration Interfaces

One entity that can appear on the GD is a software *component*.

We will use the term *component* to mean a piece of software that performs some functionality. We purposely avoid using the terms "object" and "application", as the former often implies software which is passive and which encapsulates a relatively small piece of functionality, and the latter often implies a relatively complete piece of functionality. A GD component could be either, or neither, of these. Nonetheless, we will sometimes refer to a desktop component as an application when it makes sense to do so.

To the end-user, any component on the desktop appears to be a "live". This may be an illusion; some components really represent functionality which becomes activated only in response to certain end-user actions.

A component embodies some *behavior* and presents a *graphical user interface* (GUI) with which it carries out its specialized end-user interactions. In addition, a GD component may have one or more *collaboration interfaces*.

Collaboration Interfaces Collaboration interfaces represent the component's ability to engage in *collaborations* with other components. Collaborations are interactions between two collaboration interfaces via message exchange. Any type of information may be exchanged during a collaboration, including desktop entities themselves. The rules governing this message exchange are given by a *collaboration specification*. Besides message type signatures and directionality of messages, collaboration specifications also detail legal orderings of messages. For instance, a bidder component cannot send an auction component a bid before it has registered with the auction. The two collaboration interfaces participating in a given collaboration play two "roles" defined by the collaboration specification. We call these roles "plug" and "socket," reflecting the way collaboration interfaces are visualized.[1]

A collaboration interface of a component represents the ability of that component to play a particular role in a particular collaboration, and thereby to participate with any other component that can play the opposite role in the same collaboration. For example, a bidding application might contain a collaboration interface representing its ability to play the "bidder" (socket) role in an "auction" collaboration. Such a component could potentially collaborate via this interface with any other component containing a collaboration interface for the "auctioneer" (plug) role in an "auction" collaboration. A given collaboration interface may collaborate with different partners during its lifetime.

A component may present multiple collaboration interfaces of the same type. For example, an auctioneer program may present multiple "auction plug" interfaces so it can connect to multiple bidders. A component may also add or remove collaboration interfaces as needed. For example, rather than presenting a fixed set of "auction plug" interfaces, an auctioneer program may instead produce new "auction plug" interfaces as needed to accommodate an open-ended number of bidders.

[1] In the future, the GD may include direct support for n-way collaborations; at present, n-way collaborations can be modeled via multiple two-way collaborations.

Component Visualization Components appear on the desktop as one or more windows (or icons, when iconified). Included in these windows are the elements of the component's specialized GUI. In addition, one of the windows or icons is fitted with *decorations* supplied by the desktop, including a *title bar* and a *collaboration panel*.

The title bar is similar in purpose to title bars provided by conventional window managers; it identifies the component and provides a menu for performing various desktop operations on the component.

The collaboration panel contains *component connector* icons representing collaboration interfaces currently provided by component. Each component connector is represented by one of six icons: an unattached plug, an unattached socket, an attached plug or socket, an "input slot" or an "output slot." The first four of these identify the role and state (connected or unconnected) of the collaboration interface. The meaning of the latter two icons is described below.

The plug/socket/slot icons are color-coded to represent associated collaboration types, to aid the user in identifying potentially useful collaborations.

2.1.2 Wires and Wire Connectors

A *wire* is an entity which exists for the purpose of forming collaborations, as described in section 2.2. Each end of a wire is a desktop entity called a *wire connector*. Wire connectors are displayed as plug and socket icons, attached to lines representing wires. If the two ends of a wire reside on different desktops, each will appear as the appropriate icon with its attached wire disappearing into a "hole" in its desktop.

2.1.3 Data

A *data* entity on the GD is a carrier of data. Unlike a component, it has no entity-specific GUI, has no specialized behavior, and carries no collaboration interfaces. It is visualized on the desktop via an icon selected according to its type, content, or intended use.

2.2 Creating collaborations

The end-user can inspect each component connector in the collaboration panel of a component to obtain information about the proper use of that collaboration interface. Besides technical information identifying collaboration type, each connector provides, in a user-friendly style, information describing the functionality it offers or requires. For instance, an "auction socket" connector of bidding application may display the text, "*Connect a wire from an auction component here to participate in the auction.*"

When the end-user wants to establish a particular collaboration, she "wires" together the two component connectors representing the plug and socket roles of the collaboration by dragging the mouse between the two icons. Once a collaboration is established, the participating components communicate with one another according to the rules dictated by the collaboration specification.

The fact that the two components are now engaged in a collaboration is indicated graphically to the user, by changing the appearance of the component connectors to attached plug and socket icons. Furthermore, whenever the user positions her mouse on top of one of these connectors, the other connector (in the collaboration panel of the other component) is highlighted.

Of course, the user may try to compose two component connectors that do not represent compatible collaboration interfaces. In this case, the GD indicates to the end-user that the two components cannot participate in the requested collaboration. Sometimes such failures are due to technical differences in the interfaces which can be bridged using *adaptors*. Part of our ongoing research is to examine ways that we can make use of information in the collaboration specifications to automatically (or semi-automatically) generate such adaptors.

Just as the end-user can initiate collaboration between components, she can also terminate collaborations. When the end-user tells the GD to terminate a particular collaboration (by clicking on either component connector and selecting from a pop-up menu), the GD enters into a "disconnect" negotiation with the two components to terminate the collaboration.

Wire Connectors and Distributed Access The scenario described above, in which the end-user draws a wire between two component connectors to establish a new collaboration, is really a "shorthand" for a three-step process: (1) create a new wire; (2) attach one end of the wire to one component connector; (3) attach the other end of the wire to the other component connector.

Wire connectors are especially useful when creating *distributed* collaborations between components that reside on different machines. For example, suppose Mary wishes to provide Joe access to some specific functionality provided by her database. She can create a wire and attach one end to the appropriate connector in her database component's collaboration panel. She can then drop the free wire connector into a mailer component on her desktop, and address it to Joe. Joe can later extract the wire connector from his mailbox and attach it to connectors of components (e.g. a spreadsheet) on his own desktop. The result is a distributed application created via graphical end-user operations.

A fully detached wire is completely generic, in that it can serve to create any type of collaboration. In contrast, once either end of a wire is attached, the other end is constrained by the desktop to attach only to component connectors representing matching collaboration interfaces. This is indicated by color-coding the wire connector by the same color as the attached component connector.

2.3 Dynamic Component Connectors

Though many components will be designed to support a fixed set of collaboration interfaces, components may also dynamically create and destroy collaboration interfaces. This capability can be generally useful, especially in the case of an unbounded n-way collaboration represented by multiple 2-way collaborations.

For example, an auctioneer typically wishes to collaborate with multiple bidders to form an n-way collaboration among itself and $n-1$ bidders. The auctioneer will dynamically create new "auction plug" collaboration interfaces in response to requests. For example, it may collaborate with an "auction registry" so that when a viewer of the registry indicates the desire to bid in that auction, a message is sent to the auctioneer. The auctioneer responds by creating a new collaboration interface, attaching a wire to that interface, and returning the other end of that wire to the requestor. The requestor is then free to attach that end of the wire to her bidding component, or as illustrated in the introduction, to send it in the mail to her friend.

2.4 Slots and Containment

One special sort of collaboration specification in the GD is the DTO collaboration specification, which supports the one-way transfer of DeskTop Objects. This collaboration may be used in the usual way: a component containing a "DTO plug" interface may be wired to a component containing a "DTO socket" interface to create a collaboration in which the component containing the plug may send desktop objects to the component containing the socket.

However, DTO sockets and plugs can also be transformed into *input slots* and *output slots*, respectively. Input slots and output slots allow the direct transfer of desktop entities between the desktop and the component.

When a particular DTO plug interface is "slotified", any desktop entities sent by the component to that interface are *emitted* to the desktop and will be visualized as new windows or icons. Similarly, when a particular DTO socket interface is "slotified", desktop entities may be dragged into the slot by the end user. These entities will disappear from the desktop, and will now be *contained* within the component into which they were dragged.

The appearance of contained entities depends upon how the containing component is programmed. Some containers, such as documents, will *embed* the GUI of the embedded component within their own GUI. Other containers, such as program repositories will display only the descriptions. Still others will display nothing but will perform some function when an entity is dropped into a slot: e.g. the mailer will send the entity to a recipient; the repository will respond to the dropping of a wire connector in its "search slot" by locating a component with a collaboration interface compatible with that wire connector.

Containment may be nested: if a component A contains a component B and a desktop entity C, then A is permitted to drop C into any available input slot within B. In general, a container program has the capability of performing any operation with the desktop entities it contains that the end-user can perform with the entities visible on her desktop. The programmer's view of containment is discussed in section 3.1.2.

Two consequences of the ability to move desktop entities from the desktop to a containing component, from one component to another, and then from a containing component to possibly a different desktop is that: (1) new distributed collaborations are created as a side-effect of moving entities across desktops, and (2) a component's GUI may dynamically move to a new display with a possibly different windowing system. Implementation issues regarding the latter point are discussed in section 3.1.3.

2.5 Collaboration-based search

If end-users are to effectively leverage the component-rich GD, they need new and powerful ways to find components that can be composed with existing components. The GD repository, which allows for storage and retrieval of desktop entities, is one example of a tool supporting search capabilities based on collaboration specifications.

As mentioned above, the end-user performs a search by dropping an unattached wire connector into the "search" slot of the repository. The repository then searches for components in its store that have an unattached component connector compatible with the wire connector (i.e. supporting the same collaboration specification and of opposite plug/socket polarity). The repository displays the results of the search in a menu from which the end-user can retrieve components, possibly with the dropped wire automatically attached.

3 Implementation

So far, we have described the GD from the end-user's point of view. In this section, we will briefly discuss some of the key architectural features of the GD, and also describe the GD from the programmer's point-of-view.

The architecture of the GD has been constructed with the following design goals:

- Heterogeneity: allow components written in different languages or using different distributed communication packages to collaborate on the GD;
- Simplicity: components should concentrate on application-specific logic and should be shielded from GD details (such as user requests to link to other components, rendering of component connectors in the collaboration panel, etc.);
- Transparency: components should not deal with distribution issues. In particular, a component should be insensitive to whether it is collaborating with a local or remote component.

We first discuss the architecture of the system which is visible to application programmers; then we briefly discuss the implementation.

3.1 Programmer's View

Some components will be adapted from existing code and others will be written as new code. We assume that the components being adapted to run on the GD were initially written to some model of interaction, such as DCE [9] or OLE [7].

For adapted components, simplicity for the programmer means minimizing the changes to existing code. For new components, simplicity means providing a "native" programming interface to the GD that is "thin" and provides a appropriate abstraction to the programmer. (In contrast to many environments which have "thick" interfaces and require the programmer to have intimate knowledge of the environment). We deal with both cases by augmenting the programmer-supplied code with a *wrapper*.

For many components, the programmer can be shielded from the GD and its operations, letting these operations be handled by the wrapper (see section 3.2.2). Nonetheless, the programmer still needs to declaratively identify which collaboration specifications are to be supported by the component, and to group together those collections of sending and receiving messages which constitute a single collaboration interface. This declarative specification can then be macro-expanded into an initialization section which will: (1) create a wrapper module, and (2) initialize the appropriate bindings from the application into collaboration specific interfaces in the wrapper.

Slightly more elaborate knowledge is required if the application is to be "desktop aware"—e.g. if it is going to dynamically create and destroy new collaboration interfaces, if it needs to be informed of desktop events such as wire attachments, or if its job is to manipulate desktop entities by connecting them, dropping other entities into their input slots or retrieving entities from their output slots. The wrapper provides an explicit interface to the application allowing these sorts of operations.

Additionally, if an application wants to support the capability of moving its GUI from desktop to desktop, it must either supply its own operations to unmap and remap the GUI or separate its GUI display code out into a different component. This is discussed in slightly more detail in 3.1.3, where a GD software package that makes it easier to build moveable GUIs is also discussed.

3.1.1 Programmer's View of Collaborations

In addition to shielding the programmer from many GD details, another benefit of using wrappers is that we can separate the programmer's view of a collaboration and the *contractual* view of a collaboration.

A *contract* is an abstraction representing the shared understanding of the parties to the collaboration. When two components set up a collaboration based upon a particular collaboration specification, each component can have a unique "endpoint-specific" view of the collaboration, as long as each of these views can be mapped into the same contractual view. The wrappers of the components can translate between the contractual "on-the-wire" view and the component view.[2]

This idea of separating the contractual view of a collaboration from its "endpoint-specific mapping" is adapted from our implementation language Concert/C [1, 2]. There it is used to allow different datatype representation choices (e.g. linked list vs. array) and different implementation languages at each application endpoint. Our wrappers further extend this notion to allow additional sorts of differences. For example, in the auction collabspec, the contractual collaboration specification provides simple calls to be invoked by the the bidder, such as *SendBid*, etc. The wrapper may display a slightly different interface to the bidder application, allowing such options as:

- When a bidder issues *SendBid*, the auctioneer may simultaneously be issuing *UpdateBid*, and there may be ambiguity as to which was issued first. The programmer has a choice of an "optimistic" interface in which a *SendBid* is initially accepted but may be rejected later, or a "pessimistic" one in which *SendBid* is immediately rejected if the auctioneer succeeded in issuing *UpdateBid* first. The pessimistic interface is easier to program to but may be slower since the wrappers must execute a mutual exclusion protocol "under the covers." Conversely the optimistic interface may be faster but requires the application to be able to undo bids if they are later rejected.

- The programmer can choose whether a *SendBid* call blocks or not. If not, the wrapper will need to signal the application when the response arrives. This complicates the bidder-wrapper API, but gives the writter of the bidder program the flexibility to wait for multiple events.

Part of our ongoing implementation effort is the construction of a tool which, given a collaboration specification in the form of an annotated finite-state graph, and the declarative specification of the endpoint-specific refinements, will generate automatically: (1) the text of the API (e.g. the *.h* file for a C program), and (2) the code of the compconn module for the particular collaboration interface.

3.1.2 Programmer's View of Containment

As mentioned above and in section 2.4, some components send and receive desktop entities (components, wire connectors, and data), and can be containers of these desktop entities. A container of a desktop entity is the owner of a component, wire, or data *Desktop Object (DTO)*. An application that owns a component DTO or a wire DTO can:

[2] Note that usage of these sophisticated wrappers imply that components engaged in a collaboration do not communicate directly, but through their wrappers. These wrappers can potentially be combined with marshalling code that already exists for distributed communication.

- determine the current description of a component or wire connector,
- attach or detach wire connectors to connectors on components whose DTO it also owns,
- receive "callbacks" when the state of a component or wire connector changes.

When the GD itself owns a DTO, it makes the entity visible on the desktop, and enables end-users to interact with it (see 3.2.1). Other components (such as the GD repository) may own DTOs to enable the end-user to store, search, and retrieve them, or may own them in order to ship them to other desktops (such as the GD mailer). A component which owns a DTO may or may not choose to display the GUI of that DTO within its window.

3.1.3 Programmer's View of GUIs

During its lifetime, a component may be moved from the desktop to the interior of another component (by dropping into an input slot of that component, see section 2.4). It may later be moved from component to component, and then re-emitted on a different desktop. This creates a technical problem for components written to some of the popular graphical interfaces, since these systems do not conveniently support either (1) dynamically embedding one application's GUI within the interior of another application's GUI; or (2) moving a complete GUI from one display to another. The problem is even worse in a heterogeneous environment, since it is possible for a GUI to be transferred between desktops running incompatible windowing systems (e.g. between a Unix workstation running X windows and a PS/2 running Presentation Manager).

Components that cannot redisplay on different desktops are not excluded from participation in the GD (e.g., they can still participate in collaborations and can be connected to wire connectors that are distributed to remote desktops, etc.). Components that have the ability to dynamically move from desktop to desktop must be programmed in one of the following ways:

- They accept requests to unmap and remap windows and to attach window decorations.
- They separate the component into two parts: a functional component, and a *viewer*, which communicate via a collaboration. The viewer may itself use a system-dependent interface to the display. When the component is moved off the desktop, the viewer is destroyed, and when the component is moved onto a new desktop, the new desktop searches a viewer library for an appropriate instance of a viewer, and attaches that viewer. The functional component and the viewer then interact to redisplay the GUI.

The GD software includes a window-system independent GUI package called "Movable Windows" that supports the second paradigm given above, with the added advantage of not requiring the application to redisplay newly mapped GUIs. Using Movable Windows, GUI elements are built from trees of abstract graphic objects (*GOBs*) These GOBs offer the application an abstract interface to the GUI, and they also encapsulate the state of the user interface. When a component appears on a desktop, its GOB tree is *bound* to the display associated with the desktop. This entails the creation of "concrete look-and-feel" (*CLF*) objects that realize the GUI elements encapsulated by the GOBs in a style suitable to the native windowing system. GUI elements created by the desktop itself (collaboration panels, wire connector icons, etc.) are implemented using the Movable Windows facility. Components using the Movable Windows interface to access the GUI can move across desktops fully transparently with no need for any programming to display the window or to refresh the GUI state. This is because the state of the GUI is captured, in a window-system independent fashion, in the GOBs. Note that when binding a GOB to a desktop, an appropriate CLF must still be found, similar to finding a viewer in the approach given above. However, many applications may share the same GOB abstractions and can therefore also reuse the same CLFs.

Each desktop component contains the root of the GOB tree for its display. When the component is passed to another component, the GOB tree can simply be embedded as a subtree of the containing component's GOB tree.

3.2 Implementation Technique

We will discuss three of the main modules in the GD: the *wrapper* module, the *wire* module, and the *graphical adaptor* (GA). Another important part of the architecture is the *GOB Manager*, which implements the Movable Windows architecture. The Graphical Object Manager is the subject of a separate paper, and will not be discussed further here. The basic architecture described below has been implemented in both the Hermes language [13] and the Concert/C language [1]. Differences between these implementations (due to Hermes' extremely lightweight implementation of processes in comparison to Concert/C's more traditional implementation) are described in [6].

3.2.1 The Graphical Adaptor

The Graphical Adaptor (GA) is responsible for creating and managing GD-specific GUI elements (collaboration panels, wire connectors, etc.) and responding to end-user manipulation of those GUI elements. There is one graphical adaptor per desktop.

The GA holds a set of desktop entities: either component DTOs or wire DTOs (data DTOs are treated here as degenerate cases of component DTOs). These DTOs represent entities which were created when the GA started up, emitted from output slots of other components, or otherwise sent to the GA's "emit" port.

Each time the GA receives a new DTO, it performs a *bind* operation on the DTO. This operation notifies the wrapper (for a component) or wire (for a wireconn) that this GA wants to be informed about any changes of state that may take place in the DTO. For example, a component may create a new component connector, or a wire connector's mate (the connector at the other end of the wire) may become attached.

When a DTO leaves a desktop (e.g. it is destroyed or dropped into an input slot), it is first unbound.

Once a DTO has successfully bound to a desktop, the GA creates and displays appropriate GUI elements (e.g. the title bar and collaboration panel decorating a component-supplied window, or an icon representing a wire connector). When not binding or unbinding DTO's, the GA behaves in an event-driven manner. It responds to the end-user's graphical gestures for performing desktop operations, like attaching wires and dropping components into input slots, by adjusting graphical appearances appropriately and invoking operations on the associated component wrappers and wire objects. It similarly responds to notifications of state changes from bound entities by updating or destroying associated graphical elements.

It is also possible for an application which owns DTOs to create its own "nested" GA desktop environment and perform desktop operations for its contained desktop entities. Such an application may or may not appear as a component on some other "outer" desktop, and if so, it may or may not expose the entities that appear on its "inner" desktop in the GUI it presents to the "outer" desktop.

3.2.2 The wrapper and compconn modules

A wrapper is responsible for responding, on behalf of its associated component, to desktop operations initiated by the GA. Wrapper code deals with desktop events such as attach, detach, and slotify, and negotiates at connection time with wrappers for other components to establish collaborations. As discussed in section 3.1, wrappers shield much complexity from the application and simplifies the adaptation of other applications to the GD.

A wrapper consists of a combination of generic code and code specialized for the particular collaboration specifications used by the component. Upon being launched, the wrapper creates as many of these specialized *compconn* module instances as there are collaboration interfaces supplied by the component. These specialized compconns are derived from the contractual description of the collaboration specification, plus the endpoint-specific refinement. The wrapper also responds to subsequent requests from the application to create or destroy collaboration interfaces. The wrapper remembers whether it is currently "bound" to a GA, and if so, it informs the GA of all changes to the interfaces of a component so that these changes can be reflected in the visualization of the component.

Besides responding to requests from the application, the wrapper also responds to user initiated requests (as forwarded from the GA). In most of these cases, such as when the user requests that a component interface be linked to another component interface (via a wire), the generic part simply forwards the request to the corresponding compconn module. The compconn module handles attach and detach requests from wires.

When a collaboration is established (i.e. when the second end of a wire becomes attached), the two involved compconn modules must negotiate a message exchange protocol. Each compconn module must then convert between the specific API presented to the application and that message exchange protocol. The message exchange protocol is based upon the collabspec, but in some cases may be more elaborate than the collabspec. The reason for this is that a collabspec defines a common event ordering which must be consistent on both the plug and socket side. In contention situations extra messages involving either lock acquisition or positive and negative acknowledgements must flow to ensure agreement in those cases where both partners in the collaboration are enabled to send a message. Part of our ongoing research is in automating the construction of these more complex compconns.

In our implementations of the GD, we used Hermes [13] and Concert/C [1] to implement compconns. These language implementations automatically took care of the communication and data conversion needed to send typed data across address spaces or across machines. Therefore our compconns did not have to do explicit "marshalling" and "demarshalling" of data. Without these languages or other remote communications packages, these functions would have to be performed in the compconns.

3.2.3 Wires

A wire object monitors attach and detach operations involving the two wire connectors associated with a wire, and coordinates connection and disconnection protocols between attached compconn modules. There is a wire module for each wire.

Wires respond to requests issued by the owners of either of their two wireconns. (Recall that the two ends of a wire may lie on different desktops or be owned by different components.) When a GA attaches a wireconn to a compconn, it passes the wireconn to the component's wrapper, which forwards the request to the appropriate compconn module. The compconn module calls the wire module passing it bindings to its own wiring interface (WI). These WI bindings are retained by the wire as long as it is attached. When both ends of a wire become attached, the wire has two WIs—one for the plug, and one for the socket. The wire then invokes a polymorphic connection protocol by calling *PRECONNECT* on the socket WI. PRECONNECT returns a self-describing token which is then passed to the plug compconn via the *CONNECT* operation of its WI. The plug's compconn module checks the token, and if it is the right type, it uses the token to negotiate a connection between itself and the socket's compconn module.

4 Conclusions

Language technology to support heterogeneity and distribution is valuable. But more than that is needed to empower non-programmer end-users to create, distribute, reconfigure, and embed heterogeneous components.

Increasingly, tools that present graphical metaphors are being developed to help programmers create and customize their applications. Tools such as HeNCE [3] are used to help programmers configure their software on distributed systems and a variety of GUI-builder tools help users build GUI applications. Environments such as Digitalk Parts [5] can be used in limited ways to link applications (on a per method basis). None of these, however, can be called *end-user* environments.

The goal of the Global Desktop, described in this paper, is to construct an environment that makes it easy for non-programmer end-users to link their applications together to perform a given task. It allows these end-users to access and distribute resources around a network. The key notion in our environment is component-to-component *collaboration*. The GD offers the end-user the capability to create and destroy collaborations between components, and to move components and collaborations from desktop to desktop and from one containing component to another. The GD provides a simple graphical look-and-feel for this model based upon the familiar concepts of plugs, sockets, and slots.

This work presents a number of interesting implementation challenges, including the separation of collaboration type definitions into contractual and endpoint-specific information, the generation of collaboration modules which resolve contention and convert between alternative APIs, and the development of approaches to dynamically move displays of graphical user interfaces across diverse window systems. Ongoing research is exploring tools that allow components with slightly different views of the collaboration to work together with the help of an *adaptor*.

A prototype implementation has been completed and demonstrated at OOPSLA '93 in Washington and CASCON '93 in Toronto.

References

[1] Joshua S. Auerbach, Arthur P. Goldberg, Ajei S. Gopal, Mark T. Kennedy, and James R. Russell. Concert/C: A language for distributed programming (extended abstract). Technical Report RC19044, IBM T. J. Watson Research Center, 1993. To appear in Usenix '94 Winter Conference, available from the authors.

[2] Joshua S. Auerbach and James R. Russell. The Concert Signature Representation: IDL as intermediate language. Technical Report RC19229, IBM T. J. Watson Research Center, 1993. To appear the 1994 ACM SIGPLAN Workshop on Interface Definition Languages.

[3] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Supercomputing 91*, 1991.

[4] M. Cagan. The HP softbench environment: An architecture for a new generation of software tools. *Hewlett-Packard Journal*, 41(3), June 1990.

[5] Digitalk. *PARTS Workbench User's Guide*, February 1993.

[6] A. Lowry, R. Strom, D. Yellin, and M. Raghavachari. Experience with and without featherweight processes in building a distributed middleware prototype. To be available as IBM technical report, November 1993.

[7] Microsoft. *Object Linking and Programming, Version 2.0*, April 1993.

[8] Object Management Group and X/Open. *The Common Object Request Broker: Architecture and Specification*, 1992. Document Number 91.12.1.

[9] Open Software Foundation, Cambridge, Mass. *OSF DCE Release 1.0 Developer's Kit Documentation Set*, February 1991.

[10] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6:307–334, 1986.

[11] J. Purtilo. The polylith software bus. Technical Report TR-2469, University of Maryland, 1991.

[12] S. Reiss. Connecting tools using message passing. *IEEE Software*, 7(4), July 1990.

[13] Robert E. Strom, David F. Bacon, Arthur Goldberg, Andy Lowry, Daniel Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing.* Prentice Hall, January 1991.

[14] Sun Microsystems. *SUN Network Programming*, 1988.

[15] SunSoft, Inc., Mountain View, CA. *The ToolTalk Service*, June 1991.