OOPSLA'92
Vancouver, British Columbia, Canada
5 – 10 October 1992

Addendum to the Proceedings

Poster Submission— A Visual Environment for Distributed Object-Oriented Multi-Applications

Report by:

Robert Strom
Daniel Yellin
IBM T. J. Watson Research Center

Traditional computing environments are oriented towards standalone programs. These programs are self-contained—they interact with the user via a virtual terminal or a window, and with other applications via a centralized file system.

However, there is an increasing trend towards *multiapplications*. Multiapplications are collections of programs which interact directly with one another. For example, I may wish to connect my mail service with a filter which automatically logs my mail, and routes bills and receipts to a home finance program. Multiapplications may even be distributed over multiple machines. For example: my home finance program may be connected to my bank; I and a colleague may be collaborating on a paper, or playing chess.

There is a considerable body of work on infrastructure to facilitate creation of multiapplications by programmers: for instance, operating system support (Mach, Windows), commercial RPC packages such as DCE RPC, and high-level languages (Emerald, Hermes, Concert/C).

There is less such support for the end-user wishing to configure multiapplications. The most widely available systems are based on Object Linking and Embedding. Although this is a useful step, it is still derived from the paradigm of applications communicating via files. Highly interactive applications, such as queries, process control, real-time music "jamming," etc., require other interaction paradigms. These paradigms are supported at the low-level for programmers, but not for end-users.

Another reason to support new interaction paradigms is the desire to let users break their applications into smaller components and reconfigure them. For example, a word processor package contains components such as editors, formatters, spelling checkers, etc. A MIDI music package contains components such as sequencers, quantizers, patch editors, score editors, etc. Users may wish to break up large packages into small components in order to mix and match components from different sources. The smaller the component, the less appropriate is a file-passing paradigm, and the more appropriate an object-based paradigm based on message exchange.

This paper describes an operating environment for distributed interacting multiapplications. It incorporates an underlying object-based programming model, based on the process model of Hermes [2] and Concert/C [1]. It incorporates an end-user visualization of this model, whose look and feel is based on analogies to familiar modular equipment found in the home, such as cable boxes, cable splitters, TVs, VCRS, stereos, etc.

The computation model is as follows: The computational objects are called *components.* Components may own data, and execute programs. The programs manipulate the local data, and exchange messages with other components via *connectors.* Connectors have an *interface type,* which defines the protocol (data type, and sequence) of messages sent by the component and expected from other components. Components are implemented by the *processes* of languages such as Hermes and Concert/C; connectors by collections of *ports.*

Each component is either owned by a *desktop* in the network, or by another component. When a component is on a desktop, it is visualized as a window or icon. The connectors are visualized as decorations called *plugs, sockets* and *slots.* Plugs and sockets on a desktop may be interconnected with *wires.* The shape and/or color of plugs, sockets, slots and wires reflects their interface type. Wires may cross desktops: such wires will appear to vanish into a "hole" in the desktop[1].

Each window on the desktop has its own graphical user interface whose appearance and function is application-specific. The appearance and function of the decorations, however, is uniform across the desktop. The following operations are supported:

- Wiring Plugs and Sockets: The user can make and break connections between plugs and sockets of matching interface type by clicking and dragging. This causes bindings to be established between the corresponding ports in the underlying programs, allowing them to communicate.
- Dropping Objects into Slots: The user can drop either a component or an unattached wire connector into a slot. This transfers the ownership of the component or of the wire connector from the desktop to the *receiving component*—the component owning the slot. The object which was dropped disappears from the desktop. The capability to manipulate the component or to connect the wire is now transferred from the end user to the program in the receiving component.
- Receiving Objects from Slots: A component may on its own initiative *emit* a component or wire connector from an output slot. This is the reverse of dropping: the ownership of the object is transferred from the emitting component to the end user at the desktop; the formerly invisible object becomes visible. Notice in particular that a distributed connection can be achieved by dropping a component or wire into a mailbox component on one desktop and having it re-emitted from a mailbox component on another desktop.
- Packaging and Unpackaging: The user can change the level of granularity at which he or she views components or connectors. A collection of components can be *packaged* into a single component. The user may specify that certain windows and certain connections of the package be hidden. Once packaged, the new composite component is manipulated (dropped, emitted, killed, etc.) as a single unit on the desktop. Unpackaging is the reverse operation which reveals the previously hidden structure of a composite component and allows the individual parts to be reconnected, dropped, destroyed, etc. Connectors may also be packaged: for example, a collection of two plugs and a socket may be grouped to form a three-pin connector.

[1] To avoid cluttering the desktop, the plugs, sockets, and wires may be hidden away when the user is not interested in examining or changing the configuration.

References

(1) Joshua Auerbach, Mark Kennedy, Jim Russell, and Shaula Yemini. "Interprocess communication in concert/C." Technical Report RC17341, IBM T. J. Watson Research Center, October 1991.

(2) Robert E. Strom, David P. Bacon, Arthur Goldberg, Andy Lowry, Daniel Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing.* Prentice Hall, January 1991.

Contact information:

Robert Strom
strom@watson.ibm.com

Daniel Yellin
dmy@watson.ibm.com

IBM T. J. Watson Research Center
PO Box 704
Yorktown Heights, NY 10598