THE DESIGN OF A POLICY-FREE, PARALLEL CORE FOR GRAPHICAL USER INTERFACE TOOLKITS

by

Joseph A. Konstan and Lawrence A. Rowe

Memorandum No. UCB/ERL M91/111

12 December 1991

THE DESIGN OF A POLICY-FREE, PARALLEL CORE FOR GRAPHICAL USER INTERFACE TOOLKITS

by

Joseph A. Konstan and Lawrence A. Rowe

Memorandum No. UCB/ERL M91/111

12 December 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

THE DESIGN OF A POLICY-FREE, PARALLEL CORE FOR GRAPHICAL USER INTERFACE TOOLKITS

by

Joseph A. Konstan and Lawrence A. Rowe

Memorandum No. UCB/ERL M91/111

12 December 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The Design of a Policy-Free, Parallel Core for Graphical User Interface Toolkits†

Joseph A. Konstan and Lawrence A. Rowe
Computer Science Division
University of California
Berkeley, California 94720

konstan@cs.berkeley.edu and rowe@cs.berkeley.edu

Abstract

This paper presents a design for an event-processing core to be used for constructing user interface toolkits. This core is designed to support concurrent execution on shared-memory multiprocessors and provides support for traditional event dispatching, geometry management, data propagation, modal event processing, form behaviors, and interprocess communication. The core is policy-free, allowing the toolkit designer to set both the programmer interface and the toolkit policies while benefiting from concurrent execution.

Keywords

User Interface Toolkits, Parallel Processing, Data Propagation, Geometry Management, Inter-Application Communication, Event Processing

Introduction

Building graphical user interface toolkits is hard.

This paper presents the design of a toolkit core that provides support for many of the more difficult parts of toolkit construction. This design is presently being implemented and we will soon proceed to build a graphical user interface toolkit using it. The rest of this section discusses the difficulties facing toolkit writers and the solutions we plan to offer them.

Graphical user interfaces, by their nature, must be high-performance. As the complexity of interfaces increases with greater use of larger displays and color and the inclusion of multimedia data, it is becoming even more essential to attain high performance to provide a responsive interface. Shared memory multiprocessors are now available for desktop workstations and we expect them to become even more economical in the coming years. Programming in parallel is difficult, however, and introduces potential race conditions and deadlocks that render an application useless or harmful. Our first goal, therefore, is to provide a core for toolkits that does not require the toolkit-writer to write parallel code to reap the benefits of concurrency.

Within toolkits, one of the more complicated features to implement is the layout and sizing of windows within a parent. This problem, known as geometry management, becomes more difficult to solve as new display types are created (e.g., video that can only be displayed in certain sized windows). Geometry management is primarily a problem in communication among the windows being managed and the agent allocating space among them. The geometry management policies are neatly isolated within the algorithms defined by this agent. Our second goal is to provide complete support for implementing any policy of geometry management by providing abstractions for communication between windows and geometry management agents.

Graphical applications tend to display multiple views upon data. A major part of writing such applications is ensuring a consistent display of data. In addition, many applications use values that are computed from other data in the application (such as graphics that are computed from a list of objects). Both of these types of data management can be more easily handled using a system of data propagation. Our third goal, therefore, is to provide a data propagation and constraint system that can be used both for building toolkits and for building applications on top of toolkits.

Many toolkit applications have a sequential feel because the toolkit is unable to process any additional events when a dialog window is active. This leads to a particularly bad interface when the dialog is prompting for information which should be derived from the other windows (e.g., a dialog in a mail program asking for the recipient often does not allow the user to view the incoming mail to search for the recipient's address). While some of these cases are a result of poor application design, many more are due to the difficulty in implementing partial modes (states where only certain operations are valid) in the underlying toolkit. Our fourth goal, therefore, is to provide support for a model of event processing that simplifies specifying and implementing these partial modes.

Computer applications were at one time visually based on the paper forms they were meant to replace. Even many of the earliest full-screen applications supported form behaviors including focusing keyboard input and support for tabbing among fields. More recent mouse-based applications have often sacrificed this form of interaction for the simpler

†This research was supported by the National Science Foundation (grants DCR-85-07256 and MIP-90-14940). Joseph Konstan was also supported by a National Defense Science and Engineering Graduate Fellowship granted through DARPA.

"type at the mouse" protocol. Many users express a preference for form-like applications because they can use the application without moving their hands from the keyboard. Indeed, many popular applications have extended form behaviors to support keyboard selection among menus and buttons. Since this is the case, our fifth goal is to provide an event processor that supports form behaviors.

Graphical programs share resources and often data. Even at the simplest level, two different programs compete for screen space and for colors (on color-mapped displays). Better coordinated applications may recognize that they are displaying the same data or that they are sharing window system resources. At present, there is very little communication among applications. Our final goal, therefore, is to support communication between applications.

The next section presents a model of event processing that the foundation on which these features are implemented. We then discuss our design for using event processing to satisfy each of these goals. Finally, we discuss related work and our plans to implement this design.

Event Processing

The core of most graphical user interfaces is an event processing loop. At its simplest, this loop reduces to:

```
event ev;
while (true) {ev = get-next-event();
              dispatch(ev); }
```

In most toolkits, the dispatch involves looking up the window in which an event occurred and the type of event (e.g., button click or control-key press) and calling the appropriate handler procedure. When windows are created, they register handlers for the events they want to receive. In general, events that are ignored by a window can be handled by a parent window instead. Some toolkits provide additional event types (such as timer events or program-generated events) which are similarly dispatched to windows.

We propose to extend this model of event processing to incorporate an extensible event type and a wider range of event recipients. The remainder of this section describes this event processing model in more detail.

It is convenient to break a user-interface system into three parts: event generators, event consumers, and the event processor. Event generators include window system servers, software and hardware timers and signals, database alerters, equipment monitors and alarms, and any other software or hardware that can generate asynchronous requests. Event consumers include windows (and toolkit widgets), data variables (that respond to database requests), and processing agents (e.g., redraw processor). Many entities in a toolkit are both generators and consumers (e.g., a window may be an event generator when it signals a geometry management request to change size and a consumer when processing the requests of its child windows). The role of the event processor is to collect events from the generators and dispatch them to the consumers.

Figure 1 illustrates the architecture of a user interface system. Queues of events connect the event generators with the processor and the processor with the consumers. In between, event processing includes determining the recipients of each event (this architecture allows for broadcast events as well as single and multiple recipient events), event consolidation (distributing multiple events as a single

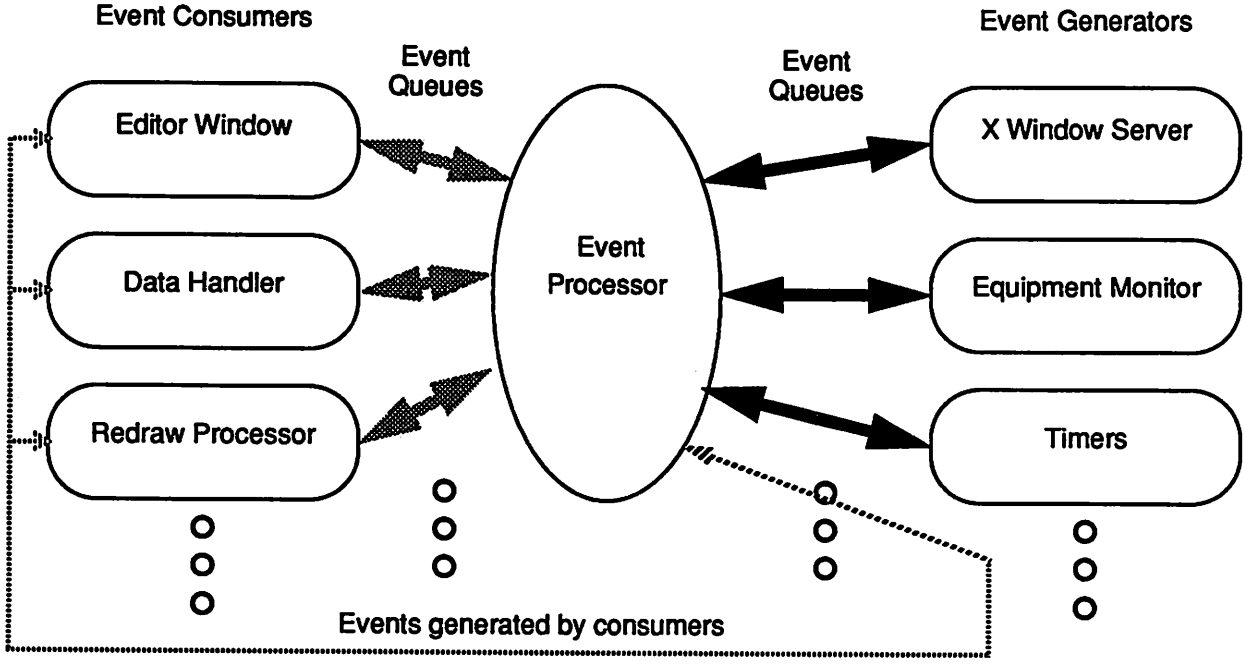

Figure 1. Event Processing Model

composite event), and synchronization and control. This processing is table driven and new event types and dispatch rules can be added at any time.

The illustration also shows a process architecture not commonly used for user interface applications. By defining a separate redraw processor, a toolkit can assure that simple redraw operations (where no data has changed, e.g., when a window is exposed because an occluding window is moved) can be processed by a single redraw handler rather than individually by each window. This redraw handler, which must have access to a copy of the desired image, can more efficiently repaint screen regions that overlap several windows. Similarly, a single data handler can assure that all data change operations occur atomically. If several displayed objects change in response to a change in data, this data handler can assure that they are displayed with either all the old values or all the new values. Current systems, such as Tk [9], implement this type of transaction system by delaying in hopes of receiving all data changes before redrawing the display. The data handler approach allows explicit transaction control events to be processed.

This general design does not address issues of event priority or order of processing. These decisions are deliberately left open to the toolkit writer. When defining event types, the toolkit designer may specify any processing style desired (including looking ahead for other related events). The relative priorities of different event types is also specified by the designer.

This event processing model is a configurable, extensible abstraction for communicating among parts of an application. The next section describes how this architecture is implemented on a parallel processor. The following sections show how it is used to implement geometry management, data propagation, partial modes, form behaviors, and inter-application communication.

Deriving Parallelism from Event Processing

The event processing model shown in figure 1 consists of an event processor that receives events from various event generators and dispatches them to various event consumers.This model can be implemented to allow for the concurrent execution of the event generators, the event processor itself, and the event consumers with synchronization being handled by the event system.

At the simplest level it is easy to find concurrency in any event-based system. Each event generator can (and probably should) run in parallel since blocks in a single event generator should not prevent other events from being processed. Sequentiality could be re-established at the event processor by carefully reading the event queues in an appropriate order (perhaps via timestamp). The asynchronous nature of event generation assures, however, that any nondeterminacy across different event queues could have occurred in a sequential system as well.

While concurrent event generators are simple, they do not yield a large performance gain. In typical user interface applications, the packaging of events consumes only a tiny fraction of the time spent processing the events and the hard work involved in event generation is performed in separate processes (e.g., the window, database, or equipment server). We expect event packaging overhead to remain small and therefore assert that for parallel implementations to yield substantial performance benefits there must be parallelism in the event consumers.

In many applications the potential parallelism is obvious. An application with multiple windows (accessing different data) can easily process events in different windows in parallel. Requests to repaint windows, as mentioned above, can also be handled in parallel with other operations. The limits of such parallelism are not nearly as obvious. A simple example where concurrent execution is wrong involves dialog boxes. Most toolkit designers wish to impose modality with dialog boxes and therefore must prevent other windows from receiving and processing some events while the dialog is active. Whether the scope of the dialog is an entire application or the window from which it was called (or some other scope) is left, or course, to the toolkit designer.

This issue is addressed by allowing event consumers to be executed in parallel while providing the toolkit designer with ways to limit that concurrency. One way of limiting concurrency is the synchronization event. This event is intended to be dispatched to several consumers at once. It may contain other events that indicate the actions to be taken when processing (including wait for another synchronization event). The dispatcher guarantees that no consumer will receive the synchronization event until all are ready to process it and that no consumer being synchronized will receive another event until it has processed the synchronization event. An example of using a synchronization event might be changing a major mode in an application. It is important that all parts of the application change the mode at the same time to ensure that the user perceives a single mode change and that user inputs are consistently interpreted.

Another form of controlling concurrency involves changing the event dispatch tables. The event processor is table driven and therefore can be instructed to change its dispatching strategy by providing an alternate dispatch table. A modal menu or dialog box might provide a dispatch table that redirects all user inputs to itself. An application monitor might provide a dispatch table that directs a copy of each event to a log and another copy to the usual recipient. Since the timing of swapping dispatch tables is critical, it becomes essential for the event processor to act as an event recipient as well. It can receive synchronization events which it uses to control the timing of event dispatch table changes.

The remaining concurrency control primitives involve locks and interprocess communications. We plan to provide both simple locks and higher level mailboxes (in the spirit of SPUR Lisp [19]) to support other synchronization.

Geometry Management

Geometry management is the assignment of positions and sizes to windows within an application. Most toolkits have accepted the idea that windows should only make requests for sizes and locations and that a higher-level window (or its agent, a geometry manager) should be responsible for the actual layout. This process is generally recursive with the higher-level window requesting a new space allocation

for itself from above.

Geometry management can be divided into two parts: layout algorithms and window requests. Window requests specify the desired size and location for a window, acceptable size ranges and ratios, and layout-specific requests such as being placed to one side of another window or being arranged before another window. For example, a text window might request that its height be a multiple of its font height (for a whole number of lines) while a video window might need to be exactly 640 by 480 pixels because of hardware constraints.

Layout algorithms compute a layout given the present layout of a window (if any), the size of the window, and the requests from all subwindows. Many layout strategies have been used in graphical toolkits including perpendicular packing around a cavity (e.g., Tk), "rubber sheet" layout, boxes and glue layout (e.g., Interviews [6]), matrix layout, stack layout (e.g., for icon palates), and edge-displacement/stretch constraints (e.g., Picasso [15] and Next [8]). Even layouts that look identical may behave differently when resized. For instance, figure 2 shows how two geometry managers might resize subwindows when the main window is enlarged. The rubber sheet geometry manager expands each window equally (as if the windows were literally stretched on a rubber sheet) while the cavity pack geometry manager maximizes the amount of space allocated to the main window.

The toolkit core provides a geometry management support system that allows any layout algorithm to be supported. Indeed, although the examples assume that parent windows are responsible for managing their children, we intend to support systems where children have control of their own sizes and locations or where the geometry management hierarchy is separate from the window tree.

There are three essential features in this geometry management support system.

- Each window being managed must completely specify a request that is independent of its present location or size
- Each layout algorithm is broken into at least two components--one that makes a request for size to the parent and one that lays out the children given the window's size allocation.
- The geometry management processing is performed in a tree-walk fashion that guarantees that each layout is performed the minimum number of times.

The rest of this section describes these three features in detail and describes the process of implementing specific geometry management policies on top of this support system.

The proposed system defines a behavior for manageable objects. These manageable objects have the property that they request space in a parent window. All manageable objects request space in the same way so that they can be managed by whichever layout algorithm the specific application writer deems appropriate. This request includes specific requests for a location (which may be left unspecified), a desired size, functions to compute acceptable sizes and locations, and algorithm-specific data. These request items are sufficient to provide all the information a layout algorithm needs to correctly position and size the window.

Layout behavior is defined in an object called a geometry manager. This geometry manager includes the algorithms needed to perform layout operations--one that calculates a size request and one that assigns sizes and positions to the windows it manages--and the data structures that the algorithms require.

Simple, event-based communication occurs between the geometry manager and the manageable windows. The geometry managers accept events such as "hello" from a new child, "request changed" from a child, and "size changed" from a parent while generating "request changed"

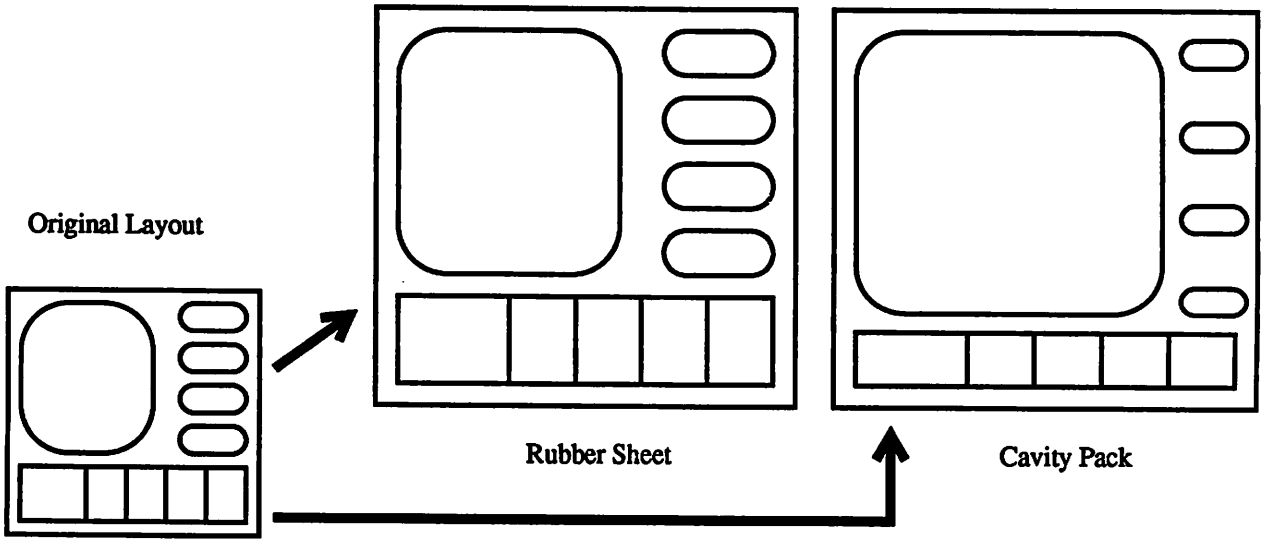

Figure 2. Examples of Different Geometry Management Layout Algorithms

messages for its parent geometry manager. The children generate "request changed," "hello," and "goodbye" messages while accepting "size changed" messages from the parent geometry manager. The toolkit core provides translations between these events and specified functions and actions so a window that changes its request automatically generates the appropriate event. Similarly, "request changed" events are translated into calls on the layout routines in the geometry manager.

The final feature of this geometry management support system is a tree-walk evaluation system that guarantees a minimum amount of layout computation. At geometry management time--typically when the user should notice any changes--if any geometry management events are pending then the geometry management process is performed in two steps. First, working from the leaves of the window tree towards the root, each geometry manager that has pending events calculates the size it wishes to request from its parent. Second, starting from the root down to the leaves, each geometry manager that has either a new size or has new events performs a layout of its children. Since this process works down the tree, each geometry manager is assured of receiving any response to its new request before performing the layout. In addition, this evaluation order creates a great deal of concurrency since siblings can be processed in parallel in each phase.

A prototype geometry management system based on an earlier version of these concepts was implemented in the Picasso system. Our experiences with that systems showed that geometry managers were quite easy to define (since only two functions needed to be specified) but they were not easy to refine or to optimize. This earlier system lacked the evaluation ordering, which accounts for much of the performance problem, but there are still cases where incremental performance tuning is desirable. As a result, we also will provide event-processing hooks for the geometry managers that permit the identification of low-computation cases (e.g., size change with no child changes, only one child changed, etc.). These special-purpose algorithms can be used whenever the event queue for a geometry manager satisfies a particular predicate.

In practice, then, writing a geometry manager involves specifying two algorithms and observing the quality and speed of the layout. The geometry manager can be optimized by adding caches or special-purpose algorithms for common cases. Throughout this process, the windows being managed need not change at all, since their communication is via a specified common protocol. The support system will provide processing that takes advantage of the parallelism inherent in a window tree with localized dependencies.

Data Propagation

Data propagation is the process of assuring that changes in data values are reflected in both the displays of those values and other related values. For example, in an application that displays shapes both graphically and textually (e.g., by showing their location and size numerically), a user action that changes the width of a shape (either numerically or graphically) should cause the other display to change and should change the underlying shape representation. While a few user interface toolkits (including Garnet [3][7], Grow [1], and Picasso[15]) have implemented data propagation systems, these systems are hard to implement and are therefore missing from most toolkits.

The support for data propagation must be defined at three levels. First, there is the definition of the propagation rules themselves (including what data changes trigger propagation and what values should be updated). Second, there is the mechanism for identifying that a data value has changed. Third, there is an internal system for actually propagating the data changes.

The definition of propagation rules is a problem that should be left to the toolkit designer. Each toolkit will have a different concept of the proper role of propagation. Some will choose to use it only for internal implementations of such data consistencies as synchronizing a scroll bar with the region it scrolls. Others will provide it only as an external resource for user applications. Many, we hope, will use it in both ways. In a the Picasso propagation system written in Common Lisp [18], propagations are defined using syntax similar to the Lisp `setf` and `let` special forms. The propagation system is by application programs and to implement the application framework (e.g., for parameter passing and some inter-window communications) [5].

The identification of changed data values is a programming language specific problem. We intend to provide appropriate mechanisms for each language as we provide the toolkit core. For Common Lisp, the language for our first prototype, we have chosen to define propagating values as slots in Common Lisp Object System objects [4]. CLOS provides efficient ways detect the change of a slot and to take actions based on that change. For other languages we might find it necessary to use a data-value setting function similar to that used in Garnet (which is written in Lisp but does not use CLOS objects). For each language, it will be clear how to change a data value so as to trigger propagation and how to change a data value clandestinely (a feature occasionally needed by toolkit implementers).

The internal propagation system itself will be event-based. Indeed, data changes will generate events and a data propagation processor will handle these events in a transactional fashion. This evaluation mechanism will prevent loops (which are not adequately prevented in Picasso) and can lead to more efficient data updates. Since this handler is provided in the base system, toolkit implementers need not worry about synchronizing data propagation with other parts of the toolkit--that will be taken care of by the base system. In addition, while the actual data updates must be executed as a critical section, the precomputation of data changes can be done in parallel.

The toolkit implementer, therefore, can use data propagations directly merely by making calls on this part of the toolkit core. The propagation system can be made available to application writers by defining a syntax appropriate for propagation rules. To simplify this task, we will provide a simple syntax for each programming language. The propagation system exploits concurrency by precomputing data changes in parallel with other execution but maintains correctness by locking out other activity when actually performing the updates.

Handling Modality

The power of window-based applications is the flexibility they offer users. Users are normally permitted to select operations and perform them in an order of their own choosing. Certain operations are inherently modal (e.g., menu operations), however, and others require immediate attention (e.g., dialogs and alerters) and therefore impose some modality by locking out other operations. User interface toolkits generally provide a separate event handler for handling menus and dialogs that only accept certain events (namely those that the menu or dialog can use) and reject any others either silently or with a warning bell or flash. In the extreme, other windows are not even allowed to repaint their contents until the user has finished a modal action (though fortunately many toolkits and applications explicitly allow repaint events to be processed). We believe that this type of all-or-nothing event processing is inadequate for many applications. This section describes a system to better support modal interactions.

The essence of the modality problem is captured by the confirmation request dialog box. This dialog, which is asking a user to confirm an operation before continuing, clearly must prevent further conflicting operations from being performed until the user replies. For instance, if the user is asked to confirm an irreversible operation on an editor, it is clear that the user should not be allowed to further edit until that operation is completed or aborted.

The conventional solution is to direct all events to the dialog which can then ignore the events unrelated to its own operation. This solution solves the immediate problem but does not allow the user to continue to work in other windows, to see the entire contents of the window affected by the operation, or even to consult the help system to learn about the operation being confirmed! Ignoring all unrelated events is clearly a poor design. To help solve this we will support dialogs and other modal interactors that can specify changes in the event dispatch tables, and accordingly be able to handle or discard certain events while allowing others to be dispatched normally.

These changes can be systematically implemented using two levels of dispatch override. At the coarse level, events of specified types can be delayed, handled, or discarded for a top-level window and all of its descendents. At the fine level, a specific event for any window can be delayed, handled, or discarded. In practice, we expect the coarse granularity to suffice for most applications (since most dialogs and other modal interactors lock out entire windows) and the fine granularity to be used when a modal interaction merely is temporarily disabling a single control or widget. Since the event types are specified as well, dialogs can permit repainting (and perhaps even asynchronous data updates) while restricting other operations.

The implementation of modality support is accomplished using the event system. Dialogs, menus, and other modal interactors can create alternate event-dispatch tables that reside in the event processor. The dialog, for instance, then sends an event to the dispatcher that activates the alternate table(s) when it is called and one that deactivates them when it returns. To combine this with window-system actions (such as grabbing a mouse), the dialog can send a synchronized event pair to the event processor and the window system. The event processor is responsible for maintaining the integrity of the dispatch tables and only allows modifications of the types listed above.

The toolkit writer, then, can provide any semantics for which actions are disallowed by a particular dialog or menu, or he can design a toolkit that leaves the decision to the application writer. In either case, the toolkit no longer requires custom event loops or other irregularities that constrain its function and complicate its design. In addition, activities not affected by the modality can continue to execute in parallel.

Form Behaviors

Form behaviors, including tabbing among fields and directing keyboard input to the currently selected field, are valuable for designing high-quality user interfaces. At the same time, these form behaviors are often difficult to implement because toolkits pass through the event processing structure of the window system rather than defining abstractions that are appropriate for applications. Since form behaviors are naturally supported in the event processor, we believe that a toolkit core should provide them to the toolkit writer.

There are many ways of implementing form behaviors. Widgets could be modified to recognize focus and tabbing characters. Higher-level windows could intercept all events and redispatch them to the appropriate child. We reject both of these techniques because they impose too great a burden on the toolkit writer. Instead, we believe that form behaviors should be implemented by providing event dispatching semantics that allow an event to be offered to several windows in succession until the appropriate recipient accepts it.

Specifically, we propose an event dispatching system where a list of windows can request any specific event. When the event comes in, the highest level (most removed from the window to which the event is directed) window is offered the event. It can process the event, discard it, or allow it to be passed on. In processing an event, a window can also redispatch it to another window for handling.

This dispatch system is similar to method combination in CLOS (using the least-specific-first combination type). It is particularly useful for implementing control-keys that perform actions far removed from the input window. This system is similar to the event model in the Andrew Toolkit [11], though we propose to follow the window tree for event dispatching rather than create a different tree structure. The benefits Andrew derives from that separate structure (e.g., allowing a scroll bar to receive events for the window it scrolls) can be achieved by having controls register for events through the nearest common ancestor.

Because of the performance-critical nature of event dispatching, these operations must be efficient. Accordingly, we will cache the list of handlers to be offered each event (which will typically be just one handler) and plan to provide opportunities for handlers to declare their intent with specific events.

In keeping with the nature of this toolkit core, no policy decisions are implemented in this form support system. Indeed, it might well be that this support is used solely for key shortcuts and not for field focus or tabbing at all. We

are already examining other models of interaction that are more easily implemented using event distribution lists (particularly hypermedia [2]). And, of course, since the support is based on event dispatching any use will benefit from the general concurrency and synchronization of the toolkit.

Inter-Application Communication

The best-used innovation provided by windowing systems is the opportunity for a user to run multiple applications on the same screen. This power should lead to a large number of smaller specialized applications just as the Unix shell led to specialized data filters. At present, the obstacle to this progression is a lack of communication among applications.

Current window systems and toolkits support very limited inter-application communication. Most window systems support a clipboard or cut buffer that can be shared between applications (e.g., the Macintosh Toolbox [13] or the X Window System [17]). Apple has further specified inter-application communications with a publish and subscribe mechanism in System 7 for the Macintosh [16]. The Tk toolkit uses the Tcl [10] `send` mechanism to implement a form of remote procedure call using the X window server to pass messages. While this mechanism is powerful, it requires defining an extensive interface to applications via the Tcl language. We plan instead to extend the event system to support inter-application events. These events could be used for both communications and synchronization.

We divide inter-application communication into several categories. First, we distinguish between applications running on the same machine in the same toolkit and those running on different machines or in different toolkits. Second, we distinguish between low-level communications (including data propagation and synchronization) and higher-level communications (the content of which is defined by the toolkit or application writer).

Applications running on the same machine in the same toolkit can be run under a single event processor. The toolkit core we have proposed easily allows the event processor to handle messages from sockets (or other input channels) and use these to spawn new event consumers and producers. For this case, all communication is provided through the shared event processor.

Applications which cannot share an event processor still communicate using events. The event dispatcher can have non-local entries in the dispatch table (which will be set up by the applications that want to communicate). All communication is in the form of events passed from one dispatcher to another (the form and medium of transport is immaterial). Indeed, a protocol should be defined to pass events between the event processors of different programming language implementations of the toolkit core.

Many of these events will be routine. Data propagation events, for instance, can be sent to share data updates between applications. For practical purposes, these can be sent in a synchronized or non-synchronized fashion. Synchronization is performed using two-phase synchronization events among the affected event processors (with the initiator being in control). Higher-level events will be sent to advertised handler names and can contain any meaningful data that can be passed between machines (i.e., no memory pointers).

We recognize that no toolkit can, by merely providing support for inter-application communication, induce a major change towards cooperating applications. The key problem is that existing applications and new applications written in other toolkits cannot take advantage of these features. To help overcome this obstacle, we plan to include a translation option for non-local events. This will allow outgoing events to be translated to formats used by other systems and incoming events to be received from other systems to be converted to our format. While this is still no panacea (since few systems have any such communication ability) we expect this translation feature to become more useful in the future and to serve as a model for programming-language independent events.

Status and Plans for Future Work

Many of the ideas presented in this paper are extensions of ideas prototyped in the Picasso toolkit and application framework. Specifically, Picasso has a rather extensive implementation of data propagation and a high-function though low-performance version of geometry management. Part of the motivation behind this proposal is the difficulty we faced developing Picasso and the desire to attain better performance by incorporating parallelism in the core. We also found that we were regularly extending our event processor to handle new event types (most notably messages from equipment monitors [14]) and special dispatching situations (for dialogs and menus).

Other researchers have faced these same issues. Data propagation (in the form of a constraint network) is an integral part of the Garnet toolkit. The Andrew toolkit implemented tree-dispatched events to handle form-related behaviors. The X Toolkit [12] specifies that geometry managers should be defined for different layout strategies and the Tk toolkit supports interprocess communication through sending Tcl commands. Each of these implementations is somewhat successful, but each is implemented in its own way, and none of these features can be easily made to work together.

We are proposing, therefore, to start at a very low level. We are building this toolkit core starting with abstractions for event processing in a parallel environment (which we also plan to implement using threads on a uniprocessor). We will then add the abstractions for data propagation and geometry management and build a small toolkit on top of this to tune the core. Finally, we will add in the more advanced event-processing features for modal interaction, form behaviors, and interprocess communication. Along the way, we will continue to develop a toolkit on top of the core, and expect it to serve not only for writing programs but also as a model for other toolkit writers using this core.

Our initial implementation will be in Common Lisp both because of out experience developing user interface systems in that environment and because of the support the Common Lisp Object System provides. Once this initial version is complete we will examine the results to determine whether a C++ version is appropriate.

Acknowledgments

We would like to thank many people without whose support this work would not be possible. We especially thank the entire Picasso development group and our initial users for feedback on these ideas and insights into implementing them. We'd specifically like to that Brian Smith who not only designed much of Picasso but also has been an invaluable source of advice on future research directions in user interface toolkits. Finally, we'd like to thank John Ousterhout, who designed the sX and Tk systems for his input and added perspective.

References

[1] P. S. Barth, "An Object-Oriented Approach to Graphical Interfaces", *ACM Transactions on Graphics, 5*, 2, April 1986.

[2] B. S. Becker and L. A. Rowe, "HIP: A Hypermedia Extension of the PICASSO Application Framework", to appear in *Proc. NIST Advanced Information Interfaces: Making Data Accessible 1991*.

[3] D. Giuse, "KR: Constraint-Based Knowledge Representation" in *The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp* (B. Myers et. al., ed.). CMU Technical Report CS-90-117, March 1990.

[4] S. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1988.

[5] J. A. Konstan and L. A. Rowe, "Developing a GUIDE Using Object-Oriented Programming", *Proc. OOPSLA '91*, Phoenix, AZ, Oct. 1991.

[6] M. A. Linton, "Composing User Interfaces with InterViews", *IEEE Computer*, Feb. 1989.

[7] B. Myers, et. al., *The Garnet Toolkit Reference Manuals: Support for Highly Interactive, Graphical User Interfaces in Lisp*, Technical Report CMU-CS-89-196, Pittsburgh, PA, Nov. 1989.

[8] Next Corporation.

[9] J. K. Ousterhout, "An X11 Toolkit Based on the Tcl Language", *USENIX Conference Proceedings*, Winter 1991.

[10] J. K. Ousterhout, "Tcl: An Embeddable Command Language", *USENIX Conference Proceedings*, Winter 1990.

[11] A. Palay, et. al., *The Andrew Toolkit--An Overview*, Information Technology Center, Carnegie-Mellon University, Pittsburgh, PA.

[12] R. Rao and S. Wallace, "The X Toolkit: The Standard Toolkit for X Version 11", *USENIX Conference Proceedings*, Summer 1987.

[13] C. Rose, et. al., *Inside Macintosh*, Addison-Wesley, Reading Mass., 1988.

[14] L. A. Rowe, et. al., *Berkeley IC-CIM Research* (videotape), University of California, Berkeley, August 1991.

[15] L. A. Rowe, J. A. Konstan, et. al. "This Picasso Application Framework", *Proc. UIST '91*, Hilton Head, S.C., November 1991.

[16] C. Rubin, *The Macintosh Bible Guide to System 7*, Goldstein & Blair, Berkeley, CA, 1991.

[17] R. W. Scheifler and J. Gettys, "The X Window System", *ACM Trans. on Graphics, 5*, 2 (Apr. 1986).

[18] G. L. Steele, *Common Lisp The Language, Second Edition*, Digital Press, 1990.

[19] B. Zorn, et. al., "MultiProcessing Extensions in SPUR Lisp", *IEEE Software, 6*, 4, July 1989.