The following paper was originally published in the

Proceedings of the 3rd USENIX Windows NT Symposium

Seattle, Washington, USA, July 12–13, 1999

HIGHER-ORDER CONCURRENT WIN32 PROGRAMMING

Riccardo Pucella

Higher-Order Concurrent Win32 Programming

Riccardo Pucella
Bell Laboratories
Lucent Technologies
riccardo@research.bell-labs.com

Abstract

We present a concurrent framework for Win32 programming based on Concurrent ML, a concurrent language with higher-order functions, static typing, lightweight threads and synchronous communication channels. The key points of the framework are the move from an event loop model to a threaded model for the processing of window messages, and the decoupling of controls notifications from the system messages. This last point allows us to derive a general way of writing controls that leads to easy composition, and can accommodate ActiveX Controls in a transparent way.

1 Introduction

Programming user interfaces on the Windows operating system at the level of the Win32 API, its lowest level, is hard under the best conditions and maddening the rest of the time. At least three points underlie the difficulty:

- the API is large (more than a thousand functions) and growing;
- the system is input-driven, making it difficult to perform intensive computations and remain interactive;
- the system is centered around an event loop, yielding control-flow that is hard to understand.

The first problem is hard to circumvent, considering that the API is large because it covers a lot of functionality required by applications. However, a better structuring can help reduce the burden of using the API, as demonstrated by the popularity of C++ class frameworks such as *Microsoft Foundation Classes* (MFC) or Borland's *Object Windows Library* (OWL). The remaining two problems are in fact related, consequences of Win32 being a message-passing API based on callback procedures [12].

It has been recognized that to ensure interactivity of applications in the presence of computation-intensive code, multiple threads should be used [2, 5]. It turns out that by using a truly concurrent approach rather than the system-level threads available from the NT kernel, one can construct a framework that also solves the last problem, the event loop control-flow nightmare [15, 7, 5].

EXene [6] is a user interface toolkit for the X Windows windowing system [20], built on top of Concurrent ML, a concurrent language providing higher-order functions, static typing, lightweight threads and synchronous communication channels [18]. EXene uses a threading model from the grounds up, leading to a design both elegant and simplifying a lot of the difficulties typically encountered in user interface toolkits. It is important to note that eXene interacts directly via the X protocol, without relying on an underlying toolkit.

The goal of this paper is to isolate the difficulties in providing an eXene-style framework for programming Win32 applications. We focus specifically on the two following points: moving from an event loop model to a threaded model with channel-based communication, and decoupling the handling of system messages from the interaction with controls to help implement easily composable controls. The resulting system, although fairly conservative in its abstractions, is still much simpler to use than the raw Win32 API, and further supports the thesis that moving to a framework based on a high-level concurrent language leads to a simpler system with good modularity properties.

This paper is structured as follows: after reviewing the structure of Win32 programs, we describe Concurrent ML and give an outline of the framework, focusing on the important points of window management. We then describe how controls are handled, including predefined

controls and custom controls. We also describe how to handle ActiveX Controls within the same framework in as transparent a way as possible. We conclude with some discussion of related and future work.

We assume the reader has a passing knowledge of higher-order languages in general.

2 Win32 programming

We review in this section the fundamentals of Win32 programming, at the level of tutorial books such as [14]. The Win32 API is closely linked with a given program structure. A Win32 program has an entry point `WinMain`, whose role is to initialize the application by creating the various windows making up the interface. The program then goes in a loop, reading messages from its message queue and dispatching them to the appropriate window for processing.

Classes. Every window belongs to a class, which needs to be registered prior to being used. A class sets the default icons, cursors, background colors and menu for every window of that class. A class also contains a pointer to a window procedure, which is a function invoked every time a message is sent to the window. Since a window procedure is associated with a given class, this implies that every window of that class share the same window procedure.

Window procedures. A window procedure is called whenever a message is sent to an application, either by another window or by the system. Messages are sent when a window is created, moved, resized or destroyed, when the mouse moves over the client area of the window, when the mouse buttons are clicked, when a key is pressed and the window has focus, when the window needs repainting, when a timer expires, and so on. Windows provides default handling for all of these messages, but an application will want to deal with some of them to provide its functionality.

Child windows. To simplify the creation of user interfaces, it is possible to use child windows to subdivide the area of a window into more manageable components. Each child window has its own window procedure, thereby encapsulating the behavior of the child window and allowing a certain amount of abstraction. The child window can decide to only send a few digested messages back to its parent window, which can deal with them more easily than otherwise possible. Typical example of child windows include *controls*, such as buttons, scrollbars and edit controls.

That's all there is to Win32 programming, really. Everything else is concerned with the processing of messages and their arguments, to do things such as drawing in a window (by processing the `WM_PAINT` message), handling mouse movement (by processing the `WM_MOUSEMOVE` message), handling keyboard input (by processing the `WM_KEYDOWN` message). Handling controls such as pushbuttons and edit controls is also done through messages. A pushbutton, for example, notifies its parent window of an interesting event (e.g. it has been clicked) by sending a `WM_COMMAND` message to the parent, with a code identifying the control and the notification code as arguments.

3 Concurrent ML

The language we use to express our concurrent framework is Concurrent ML (CML) [18], a concurrent extension of the mostly-functional language Standard ML (SML) [11]. SML provides among other things higher-order functions, static typing, algebraic datatypes and polymorphic types. CML is provided as a library-style extension to SML, with the following (simplified) signature:

```
structure CML : sig
  type 'a chan
  type 'a event
  type thread_id
  val channel : unit -> 'a chan
  val spawn : (unit -> unit) -> thread_id
  val sendEvt : 'a chan * 'a -> 'a event
  val recvEvt : 'a chan -> 'a event
  val wrap : 'a event * ('a -> 'b) -> 'b event
  val choose : 'a event list -> 'a event
  val sync : 'a event -> 'a
end
```

CML is based on the notion of a *thread* which is concurrently executing thread of control. A function `spawn` is used to create a thread that evaluates a given function. Communication between threads is done via *channels*. Communication is synchronous: a send blocks until a receive reads the value on the channel, and vice versa. To help design abstract communication protocols, a first-class notion of *event* is introduced. An event decouples the communication capability of an operation from its actual execution (synchronization). Sending a value over a channel is actually a two-step process: we first create an event that says that the communication operation to be done will send a value over the channel, and

when we synchronize on that event, the value is sent over the channel. Synchronization blocks until the communication is performed. Basic event constructors include `sendEvt` and `recvEvt` for sending and receiving a value over a channel. Synchronization is performed by the `sync` operation.

Decoupling definition from synchronization allows for the building of combinators to describe more refined communication mechanisms. For example, given an event, the `wrap` operation wraps a function around that event creating a new event that behaves as follows: when you synchronize on the event, the original event is synchronized on, and the result of the synchronization is fed to the function which is evaluated. Given a set of events, you can also create a new event that is a non-deterministic choice over all those events with the `choose` operation: when you synchronize on the event, one of the original events is non-deterministically chosen and synchronized on. Finally, note that channels and events are *polymorphic* over the carried type (represented by the type variable `'a`): a channel of type `int chan` carries values of type `int`, and so on.

Concurrent ML is currently distributed with the Standard ML of New Jersey compiler [1].

4 The basic framework

We outline in this section our framework for programming Win32 user interfaces using the concurrency model provided by CML. The framework is built on top of a direct binding of the Win32 API in SML. An overview of the binding as well as examples of use are given in [16]. The binding was derived from an IDL description of the API using a tool for compiling IDL descriptions to SML code interfacing the described API [17]. This turned out surprisingly well[1] and allowed us to use code found in tutorial material such as [14].

For our framework, we build on top of the raw Win32 API some layers of abstractions that simplify and abstract away from many low-level details. We still remain very much in the spirit of Win32 however, in the sense that most functions are simply lifted from the underlying API. Abstractions are mainly concerned with replacing the event loop by an independent thread and allowing a more compositional treatment of controls.

The framework aims at supporting more or less the functionality described in the first volume of [10], along with various simplifying assumptions. This paper further simplifies matters for the purpose of presentation, and in order not to overwhelm the reader with superfluous and confusing details. Note that we only give the *signature* of the modules in this paper, that is the type of the operations and values provided by the various modules. Implementation details are not discussed.

Figure 1 presents the `Run` module, which is the main entry point of the framework. The main function of a program in our framework is a function of type

```
instance -> 'a
```

taking as argument the instance handle of the program and returning some type (exactly which type is returned is unimportant). This function will be in charge of creating the various windows of the application, and calling the message loop of the main window. The function `doit` of the `Run` module invokes the main function, supplying the application instance handle.

Various modules are provided that simply encapsulate some aspect of the API, lifting the functions without trying to generalize or abstract away some of the functionality. For example, Figure 2 and 3 present modules that deal respectively with icons and menus. Other modules such as `Cursor`, `Bitmap`, `Rect`, `Pen`, `DC` encapsulate different aspects of the API. All of these are fairly straightforward, and aside from their sheer number, their implementation does not offer any difficulties. It is definitely the case that future work should aim at finding new abstractions to reduce either the size or the complexity of this part of the framework.

The most important module from our point of view is the one that focuses on *window management*. Window management describes anything that relates to the manipulation of windows, including their creation, deletion, movement, as well as the management of the classes. As we saw, every window belongs to a class, that assigns a default icon, cursor and colors for every window of that class. Moreover, in raw Win32, the class also provides a window procedure to process messages to the window. The window procedure is shared amongst all windows of the class. It is not clear why this design was chosen. Informal explanations are given that this helps guarantee that every window of a given class can behave the same way. But since the window procedure upon reception

[1] There were some interesting issues raised in providing the Win32 API bindings — both in terms of support of the Win32 API in a strongly-typed setting, and in terms of the mapping from IDL to SML — that may or may not be described in a future article.

```
structure Run : sig
  type instance
  val doit : (instance -> 'a) -> 'a
end
```

Figure 1: The `Run` module

```
structure Icon : sig
  type icon
  val application : icon
  val hand : icon
  val question : icon
  val exclamation : icon
  val asterisk : icon
  val load : Run.instance * string -> icon
  val draw : DC.hdc * int * int * icon -> unit
end
```

Figure 2: The `Icon` module

```
structure Menu : sig
  type menu
  val load : Run.instance * string -> menu
  val get : Window.window -> menu
  val create : unit -> menu
  val createPopup : unit -> menu
  val appendItem : menu * int * string -> unit
  val appendPopup : menu * menu * string -> unit
  val destroy : menu -> unit
end
```

Figure 3: The `Menu` module

```
structure Window : sig
  type class
  type window
  datatype class_style = CS_HREDRAW
                       | CS_VREDRAW
                       | ...
  datatype window_style = WS_OVERLAPPEDWINDOW
                        | ...
  datatype show_style = SW_NORMAL
                      | ...
  val class : string * Run.instance * Cursor.cursor * Icon.icon * Brush.brush * class_style list -> class
  val unregister : class -> unit
  val create : class * string * window_style list * window option * int option * int option * int option *
               int option * Menu.menu option * Run.instance * (window * Msg.msg chan -> unit) -> window
  val createChild : class * string * window_style list * window * int option * int option *
                    int option * int * int * Run.instance * (window * Msg.msg chan -> unit) -> window
  val show : window * show_style -> unit
  val update : window -> unit
  val setForeground : window -> unit
  val move : window * int * int -> unit
  val getClientRect : window -> Rect.rect
  val destroy : window -> unit
  val send : window * Msg.msg -> unit
  val quit : int -> unit
  val msg_loop : window -> int
  val default : window * Msg.msg -> unit
end
```

Figure 4: The `Window` module

of a message also receives the handle of the window to which the message is addressed, it is very easy to write a window procedure that handles messages differently depending on the recipient of the message.

In our framework, we would like to have a thread replacing the window procedure, and actual messages over channels instead of the Win32 messages passed to window procedures. In order to keep messages lightweight, we would like to drop the requirement of passing the handle of the target window when a message is sent. Indeed, our function to send a message should extract the communication channel from the window type, and send the message to that channel, implicitly determining which window the message is sent to. To help this setup, we will have a thread assigned on a *per window* basis. Of course, one can still support shared processing amongst all windows of a given class by delegating every messages to a centralized thread that communicates with every window of a class.

Figure 4 presents an excerpt of the `Window` module containing the interesting parts of the code. Types are defined for classes, windows, and various style parameters for both classes and windows. A function `class` creates a class given the appropriate parameters, and automatically registers it. The functions `create` and `createChild` are used to create windows, given the class, title, optional owner window, position and size (a value of `NONE` for these forces the use of a default, equivalent to a `CW_USEDEFAULT` in raw Win32), optional menu, instance handle and a function to process messages. This last function is spawned automatically on its own thread and is passed the window being created and a channel to communicate with the window. A child window is similar, but instead of a menu it takes an integer that should uniquely identify the child window and that will be use to communicate with the parent window. Functions are then provided to show, move and destroy the window. A function `msg_loop` is used to initiate the message loop of a window[2]. A function `send` is used to send a message to a window. The function `quit` simply posts the `WM_QUIT` message in the message queue of the application, a requirement for exiting a message loop.

As an example, consider the following main function for an application that bounces a logo around a window. This example is taken from chapter 7 of [14], and is given in its entirety in Appendix A. It is as simple an initialization function as can be: only one class, a window created with mostly default values, and a simple message loop.

[2]This assumes that windows in the framework use standard message loops, a simplifying assumption.

```
fun winmain (instance) = let
  val c = Window.class
              ("BouncingSMLN", instance,
               Cursor.arrow, Icon.application,
               Brush.white,
               [Window.CS_HREDRAW,
                Window.CS_VREDRAW])
  val w = Window.create
              (c, "Bouncing SML/NJ",
               [Window.WS_OVERLAPPEDWINDOW],
               NONE, NONE, NONE, NONE, NONE,
               NONE, instance, bounce)
  val v = Window.msg_loop (w)
in
  Window.unregister (c);
  v
end
```

The module `Msg`, outlined in Figure5, defines the various messages that can be sent to windows by the system and by other windows through the `Window.send` function. There is a datatype constructor per message, and message parameters are automatically unfolded for easy retrieval and building. The function given to `Window.create` will be spawned and passed the newly created window and a newly created channel on which the thread will receive its messages. At this point, Win32 rules for processing messages apply: every message not processed by the application must be passed to default processing, which means invoking `Window.default` with the message as argument, and so on. Often, the thread will simply read from the input channel and process the messages, but it can also listen concurrently for events coming from other parts of the application or from controls.

Finally, although we will not discuss them here, we mention that most errors in Win32 functions get mapped to SML exceptions.

5 Controls

The first step in the creation of our concurrent framework for Win32 involved lifting window procedures into actual threads with which one can communicate using CML-style message-passing. We now turn to the second important aspect of our framework: compositional controls.

A control is "... a child window an application uses in conjunction with another window to carry out simple input and output (I/O) tasks." [10]. In reality, controls can achieve any level of complexity chiefly through composition: putting a bunch of controls together forms a bigger control with potentially a higher-level semantics. It is possible in raw Win32 to compose controls, but the amount of plumbing one has to write is mind-numbing.

Our aim is to make creating new controls by combining existing ones easy, while staying within the philosophy of Win32.

A requirement for this to work is that there be no difference between a predefined control (such as a pushbutton or an edit control) and a composed control. We also would like the communication to and from the control to be independent of the window procedure of the parent window. The basic idea is that a control will have a notification channel on which it communicates internal changes and interesting events. Communication to the control is achieved by invoking appropriate functions acting on the control.

5.1 Predefined controls

Many controls are predefined in Win32. These include various kind of buttons (push, check, radio), editing controls, list and combo boxes, scrollbars, and static controls. Providing them in our framework is fundamentally a matter of presenting them the right way to the user. For example, Figure 6 and 7 give the modules implementing respectively pushbuttons and edit controls. Note the similar format of the modules: both define a type for the control, a datatype defining the various notification messages that the control can report, a CML event that a thread can synchronize on to get the notification, functions to communicate with the control, a function to create the control, and a function to access the control as a normal window, allowing one to apply functions from the `Window` module.

The problem with such an interface is that it completely contradicts the default interface for controls implemented in raw Win32. A predefined control sends notifications directly to its parent window by sending a `WM_COMMAND` message to the window procedure, with its control ID as an argument and the notification as the other. What we want is to intercept that message and redirect it onto a CML channel.

One way to achieve this is for the system to to transparently create a child window around the control, which will be the parent of the control, in charge of capturing the `WM_COMMAND` messages and sending them onto a CML channel assigned when the control is created. All very straightforward, but some work is involved in making sure that all the messages sent to the control are communicated to the transparent child window. For example, applying `Window.move` to the control should move the control but also move the transparent child window, and similarly for resizes and most other window operations.

5.2 Custom controls

Custom controls are controls defined by the programmer. To create a new control, a programmer must determine the appearance of the control and its interaction with its subcontrols, if any, and its parent. The simplest example of a custom control is a layout control, which is in charge of maintaining the layout of its subcontrols according to some constraint criterion. Other more involved controls can include dozens of subcontrols interacting in a complex way. Dialog boxes can also be seen as a type of complex control.

By uniformity, we would like custom controls to respect the informal specifications given in the previous subsection. Technically, a custom control is a child window, created via the `Window.createChild` function. The thread associated with the window, in charge of handling messages to the window, defines the appearance of the control by handling the `WM_PAINT` message, and so on. Communication with subcontrols is achieved by listening for the notification events from the subcontrols, concurrently with handling messages for the window. Similarly, a channel for reporting notification events for the custom control needs to be allocated.

For example, a new control that encapsulates two pushbuttons might have a single notification message defined as:

```
datatype notify_msg = CLICKED of int
```

which simply reports which button has been clicked, and a controlling thread processing messages to the window that also listens to notification events from the two subcontrols and sends the appropriate notification when clicks occur (assuming a notification channel `notifyCh`, and pushbuttons `b1` and `b2`):

```
...
sync (choose ([wrap (recvEvt (ch), handle_message),
               wrap (PushButton.notifyEvt (b1),
                       fn (PushButton.BN_CLICKED) =>
                             send (notifyCh,CLICKED 1)
                        | _ => ()),
               wrap (PushButton.notifyEvt (b2),
                       fn (PushButton.BN_CLICKED) =>
                             send (notifyCh,CLICKED 2)
                        | _ => ())]
...
```

```
structure Msg : sig
  datatype msg = WM_SIZE of int * int
               | WM_PAINT of Rect.rect
               | WM_DESTROY
               | WM_TIMER of int
               | ...
end
```

Figure 5: The `Msg` module

```
structure PushButton : sig
  type push_button
  datatype notify_msg = BN_CLICKED
                      | BN_DOUBLECLICKED
  val notifyEvt : push_button -> notify_msg event
  val create : string * int * int * int * int * Run.instance -> push_button
  val windowOf : push_button -> Window.window
end
```

Figure 6: The `PushButton` module

```
structure Edit : sig
  type edit
  datatype notify_msg = EN_CHANGE
                      | EN_ERRSPACE
                      | EN_HSCROLL
                      | EN_KILLFOCUS
                      | EN_MAXTEXT
                      | EN_SETFOCUS
                      | EN_UPDATE
                      | EN_VSCROLL
  val notifyEvt : edit -> notify_msg event
  val getSel : edit -> (int * int)
  val setSel : edit * int * int -> unit
  val replaceSel : edit * string -> unit
  val canUndo : edit -> bool
  val emptyUndoBuffer : edit -> unit
  val undo : edit -> unit
  val create : string * int * int * int * int * Run.instance -> edit
  val windowOf : edit -> Window.window
end
```

Figure 7: The `Edit` module

Decoupling the logic of the communication with the subcontrols from the handling of system messages to the control greatly helps modularizing the code. Indeed, given a custom control, we could easily reuse the communication logic for some other control having the same "behavior", but maybe a wildly different appearance [9].

5.3 ActiveX Controls

No discussion of controls would be up-to-date without mentioning *ActiveX Controls* [3]. The ActiveX Controls technology goes back to *Visual Basic Extensions* (VBX), a mechanism for writing control components for use in the Visual Basic environment. These were generalized to *OLE Controls* for use in a general COM-based environment [19]. The main problem with OLE Controls is that they required the programmer to implement a large number of interfaces that had to be present for the control to be usable. This did not mix well with the lightweight requirement for downloadable controls over a network, and so ActiveX Controls were introduced, fundamentally OLE Controls with looser requirements.

ActiveX Controls are simply COM objects[3], and the support for ActiveX Controls in any framework is based on the corresponding support for COM objects. An application that can use ActiveX Controls is called a *control container*. The functionality of an ActiveX Control is divided into four parts (from [3]):

- providing a user interface;
- allowing the container to invoke the control's methods;
- sending events to the container;
- learning about properties of the container's environment and allowing the control's properties to be examined and modified.

As we discussed in [16], calling the methods of a COM object from SML is fairly easy. It is harder to make the framework into a control container, because that implies presenting the whole framework as a COM object with the appropriate interfaces that ActiveX Controls can access to communicate events. This is not impossible, but most implementations of SML do not allow this to be done easily. Given a suitable implementation of such a capability, it is not hard to see how ActiveX Controls fit in the above framework. Current work on the SML/NJ runtime system is in part aimed at solving this particular problem.

[3] They must also support self-registration.

6 Related work

The idea that concurrency helps in programming user interfaces is not new. Building on the original work of Squint [15] and Montage [7], eXene [6] exemplifies the consistent use of concurrency as a foundation for user interface construction [5]. More recently, Haggis [4], a functional framework built on top of a concurrent extension to Haskell, also demonstrated the usefulness of concurrency in such a context. However, as opposed to eXene and our approach, the model presented to the user is strictly sequential — concurrency is only used internally.

Compositionality of user interface elements is a requirement for a programmer-friendly toolkit. Systems such as Tk [13] are mostly based on the notion that a user interface is a widget (in our terminology, a control) composed of subwidgets. Building a user interface is a matter of composing the controls together in a hierarchical fashion. Tk however uses Tcl as its underlying language, and because of its lack of large-scale programming structures, it is not well suited to building large systems (although some large systems have indeed been built using Tcl/Tk). The basic ideas underlying compositionality are best presented from the point of view of the so-called Model-View-Controller approach, and we refer the reader to articles such as [9] for a deeper coverage of the issues.

Of course, another closely related system is the Microsoft Foundation Classes framework, which provide C++ classes structuring most of the Win32 API. MFC also allows the definition of methods to handle messages directly, removing the need to explicitly code up the window procedure. However, the model is still based on an event loop, and it is still hard to program computation-intensive applications that remain interactive. Kernel threads must be used to help manage the complexity. More experience with our system is required before further comparison can be made, especially with respect to the efficiency, maintainability and reuse possibilities of the code.

7 Conclusion

We have described in this paper the design of a simple concurrent framework for Win32 programming, based on a high-level concurrent language with lightweight threads. The description we have given is very much an outline, and indeed even our implementation is incomplete. We have not talked about color, dialog boxes, keyboard and mouse handling, multiple-document interfaces, floating menus, common dialogs, to name a few.

The important points about our framework are the move from an event loop foundation to a threaded model, and a decoupling of the processing of system messages from the notification messages from controls. This gives us a chance to derive easily composable controls. It also gives us a natural way to incorporate ActiveX Controls transparently into the framework.

Although the framework does not introduce a great many abstractions over the underlying Win32 API, the framework is still much easier to use than a raw Win32 system, and the resulting code more modular, thereby increasing reusability.

Future work. As we mentioned, the framework is quite simplistic, and does not go as far as it could go to abstract away from the underlying system. This was an experiment to try to impose a concurrent communication mechanism onto Win32 that supports an abstract view of controls decoupled from the window procedure, and nothing else. We tried to stay as close as possible to the raw Win32 programming style. Future work is planned in two directions. First, this project is but a first step in implementing a Win32 interface to Standard ML of New Jersey. The next step is the design of a real toolkit that can manage both X-windows and Windows (and eventually others), with an even more abstract notion of controls. An investigation into the use of reactive sublanguages [8] to express the logic behind the controls interactions in such a toolkit is also in the works. Second, we plan to investigate the feasibility of transferring some of this work to a C/C++ framework, perhaps at the cost of introducing a custom version of lightweight threads.

Acknowledgments. Thanks to John Reppy for many discussions relating to the subject of concurrency in user interfaces that led to the experiment described in this paper.

Availability. The Standard ML of New Jersey distribution is available from `http://cm.bell-labs.com/cm/cs/what/smlnj`, and information on the framework presented here can be found on the author's web page at `http://cm.bell-labs.com/cm/cs/who/riccardo`.

References

[1] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Third International Symposium on Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, August 1991.

[2] J. Beveridge and R. Wiener. *Multithreading Applications in Win32*. Addison Wesley Developers Press, 1996.

[3] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[4] S. Finne and S. Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics*. Springer-Verlag, 1995.

[5] E. R. Gansner and J. H. Reppy. A foundation for user interface construction. In B. A. Myers, editor, *Languages for Developing User Interfaces*, chapter 14, pages 239–260. Jones and Bartlett Publishers, 1992.

[6] E. R. Gansner and J. H. Reppy. A multi-threaded higher-order user interface toolkit. In Bass and Dewan, editors, *User Interface Software*, volume 1 of *Software Trends*, pages 61–80. John Wiley & Sons, 1993.

[7] D. Haahr. Montage: Breaking windows into small pieces. In *Proceedings of the USENIX summer conference*, pages 289–297. USENIX, June 1990.

[8] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[9] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[10] Microsoft Corporation. *Win32 Programmer's Reference*. Microsoft Press, 1993.

[11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Mass., 1997.

[12] B. A. Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, 1991.

[13] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[14] C. Petzold. *Programming Windows 95*. Microsoft Press, 1996.

[15] R. Pike. A concurrent window system. *Computing Systems*, 2(2):133–153, 1989.

[16] R. Pucella, E. Meijer, and D. Oliva. Aspects de la programmation d'applications Win32 avec un langage fonctionnel. In *Actes des Journées Francophones des Languages Applicatifs*, pages 267–291. INRIA, 1999.

[17] R. Pucella and J. H. Reppy. An abstract IDL mapping for Standard ML, 1999. In preparation.

[18] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305. ACM Press, 1991.

[19] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[20] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

A Bounce example

```
fun bounce (window,ch) = let
  val timerID = 1
  val rate = 20
  val moveR = 10
  val xTotal = 158
  val yTotal = 131
  val xRadius = 59
  val yRadius = 45
  fun onTimer (xS,yS,xC,yC,xM,yM,b) = let
    val hdc = DC.get (window)
    val hdcMem = DC.createCompatible (hdc)
    val _ = (Bitmap.select (hdcMem,b);
             DC.bitBlt (hdc,xC - (xTotal div 2), yC - (yTotal div 2),
                        xTotal, yTotal, hdcMem,0 , 0, DC.SRCCOPY);
             DC.release (window,hdc);
             DC.delete (hdcMem))
    val xC' = xC + xM
    val yC' = yC + yM
    val xM' = if (xC' + xRadius >= xS) orelse (xC' - xRadius <= 0) then ~xM else xM
    val yM' = if (yC' + yRadius >= yS) orelse (yC' - yRadius <= 0) then ~yM else yM
  in
    (xS,yS,xC',yC',xM',yM',b)
  end
  fun computeArgs (x,y,b) = (x,y,x div 2, y div 2, moveR, moveR, b)
  fun loop (args as (xS,yS,xC,yC,xM,yM,b)) =
    case (recv (ch))
      of Msg.WM_SIZE (x,y) => loop (computeArgs (x,y,b))
       | Msg.WM_DESTROY => (Timer.kill (window,timerID);
                            Bitmap.delete (b);
                            Window.quit (window,0))
       | Msg.WM_TIMER (t) => let
            val args' = if (t=timerID) then onTimer (args) else args
          in loop (args') end
       | m => (Window.default (window,m); loop (args))
  fun init () =
    case (recv (ch))
      of Msg.WM_CREATE => (Timer.set (window,timerIR,rate,NONE);
                           loop (0,0,0,0,0,0,
                                 Bitmap.load ("smlnj.bmp")))
       | m => init ()
in
  init ()
end
fun winmain (instance) = let
  val c = Window.class ("BouncingSMLN", instance,
                        Cursor.arrow, Icon.application,
                        Brush.white,
                        [Window.CS_HREDRAW, Window.CS_VREDRAW])
  val w = Window.create (c, "Bouncing SML/NJ",
                         [Window.WS_OVERLAPPEDWINDOW],
                         NONE, NONE, NONE, NONE, NONE,
                         NONE, instance, bounce)
  val v = Window.msg_loop (w)
in
  Window.unregister (c);
  v
end
```