MrEd: An Engine for Portable Graphical User Interfaces

Matthew Flatt

Rice COMP TR96-258

March 1996

Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251-1892

MrEd: An Engine for Portable Graphical User Interfaces

Matthew Flatt

Department of Computer Science
Rice University
Houston, Texas 77005-1892
`mflatt@cs.rice.edu`

Abstract

MrEd is a new development and run-time environment for graphical user interfaces (GUIs). A MrEd programmer constructs new GUIs in Scheme, relying on Scheme's highly fexible interactive development mode for rapid prototyping. MrEd's underlying Scheme implementation provides two non-standard extensions: an object system and pre-emptive threads. Through these extensions, MrEd accomodates a rich graphics library. Since this library has been ported to the major windowing systems, GUI applications developed in MrEd immediately run on all major platforms.

1 Introduction

A GUI application is more difficult to implement than a traditional, text-based application. The additional difficulty is due to both the complexity of a GUI and the subjective quality of a GUI design. A GUI is complex because it consists of many graphical elements, each with its own state, behavior, and lifetime. Furthermore, these elements are interdependent: the state or behavior of each element can depend on the state or behavior of other elements. A GUI's subjective quality—its dependence on aesthetic and ergonomic concerns—leads to an iterative model of development in which a prototype is developed and gradually refined until the application is fully functional and comfortable to the user. This iterative development process compounds the problem of managing a GUI's complexity. A small change in the user interface design may require behavior and dependency changes to many of the graphical elements.

To address the complexity problem, traditional programming languages like C have been extended to support object-oriented programming. Using objects, a programmer organizes a program into components that are similar to the ones that an end-user sees. This method of organization is effective in managing the state and dependencies of individual graphical elements. However, the object systems of machine-oriented languages are less effective in managing element lifetimes. Although object constructors and destructors help programmers manage the explicit allocation and deallocation of graphical elements, the programmer is still responsible for correctly implementing the complex constraints that govern object lifetimes. Similarly, object systems provide some support for the iterative nature of GUI development by localizing changes, but the traditional edit-compile-link-run sequence keeps the cost of testing small changes high.

Scheme provides features that address both the complexity and iterativeness of GUI development. Scheme is easily extended with an object system and multiple threads of control. Scheme's higher-order functions provide both a powerful tool for abstraction and a lightweight alternative to objects for simple asynchronous procedure dispatching (i.e., "call-backs"). Garbage collection provides automatic management of element lifetimes. Finally, as an interactive language, Scheme supports incremental and interactive development more directly than traditional batch languages.

The number and variety of Scheme-based GUI development packages demonstrate a widespread belief in the advantages of Scheme. Nevertheless, GUI applications are still largely developed with C++, in part because existing Scheme-based tools have been inadequately engineered to handle the realities of general software development, including speed, portability, and robust integration of the language and the low-level graphics primitives. For many software projects (especially large ones), such shortcomings outweigh the benefits inherent in the development language.

MrEd is a new Scheme-based GUI development environment, suitable for developing robust and responsive GUI applications of any size. Applications that are developed in MrEd are portable among windowing systems without modification. Since text-editing, or text-like editing of more complex data, is a part of many GUI applications, MrEd provides powerful editor primitives that can be employed within a GUI application for editing plain text, editing text with multiple fonts and styles, or editing a mixture of arbitrary graphical objects (including embedded sub-editors).

Section 2 describes the architecture of MrEd. Section 3 demonstrates how applications are developed in this environment. Section 4 discusses the technical issues behind MrEd's implementation. Section 5 compares MrEd to other development environments. Section 6 presents future directions for improving MrEd.

2 MrEd

MrEd is an integrated GUI development and deployment environment. It consists of two parts: an execution kernel and a development shell. The execution kernel is a small operating system for running GUI applications; the development shell is a combined editor and command shell used to develop new kernel applications.

The execution kernel, MzScheme, is a Scheme interpreter. In addition to the standard Scheme facilities [4], MzScheme provides an object system and an extensive graphics library. This library contains primitives that define high-level GUI elements, such as windows, buttons, and menus. The graphics library is made available within Scheme through the object system, providing the graphical primitives via built-in classes. The kernel does not include a read-eval-print loop; instead, all windowing system events are passed directly to graphical objects created by applications in the kernel.

Figure 1 is a graphical overview of the execution kernel. The kernel includes a start-up module that initializes MzScheme's global environment with primitive procedures and classes. Once these primitives are installed, a designated Scheme program is loaded. This program defines the global intialization procedure `mred:startup`. Typically, invoking the `mred:startup` procedure creates one or more windows to receive input from the user. Once `mred:startup` returns, the event dispatcher takes over, dispatching events from the windowing system until the `exit` procedure is invoked.

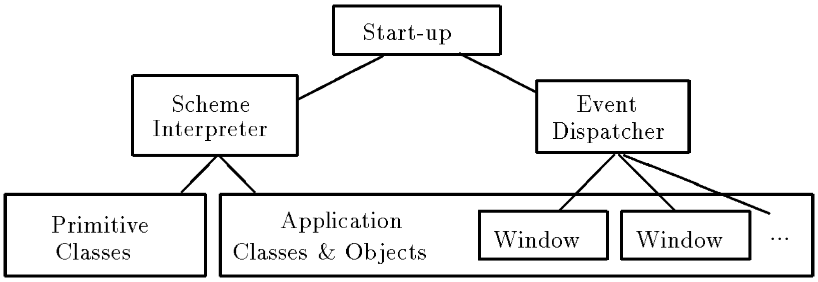

FIGURE 1: MrEd's execution kernel

A completed application requires only the execution kernel to run, but the kernel provides no mechanism for application development. Instead, MrEd includes one special GUI application, the development shell, that is loaded by default as the start-up application. The shell provides an Emacs-like editor for developing Scheme programs and testing them interactively. Because the development shell is itself a MrEd application, the shell permits extension and customization.

3 Developing MrEd Applications

To demonstrate the advantages of interactive GUI development in MrEd, we work through the development of an interactive phone book (see Figure 2). The development proceeds in three stages. First, we interactively develop the graphical elements that make up the interface, focusing on the elements' appearance rather than their function. Second, we add the search mechanism, which is triggered by callbacks connected to the graphical elements. Finally, we encapsulate the interface in a class definition.

3.1 Implementing the Interactive Phone Book

To implement the interactive phone book, we start with a frame definition:

```
(define FRAME-WIDTH 100)
(define FRAME-HEIGHT 100)
(define a-frame
  (make-object wx:frame%
              '() ; No parent frame
              "Phone Book" ; The frame's title
              -1 -1 ; Use the default position
              FRAME-WIDTH FRAME-HEIGHT))
(send a-frame show #t)
```

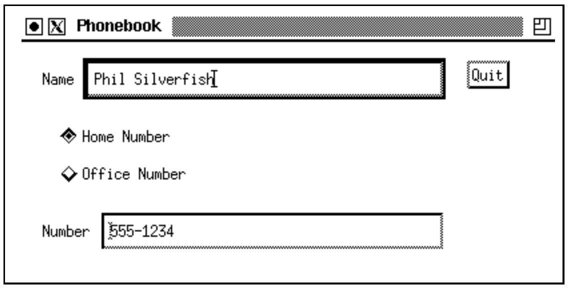

FIGURE 2: An interactive phone book interface

The procedure `make-object` takes a class and class-specific initialization parameters and returns an object that is an instance of the class. The `wx:frame%` class is the primitive frame class. The empty list is used here to indicate that the frame has no graphical parent (because it is a top-level frame). The remaining parameters specify the title, location, and size of the frame.

After this code is entered into the "phone.ss" editor buffer, a programmer may open a new console for interactively testing the program. As "phone.ss" is loaded, an empty frame appears. This new frame is the value of `a-frame`. Loading the project again creates a new frame and redefines `a-frame`. The most recently created frame can be hidden by evaluating `(send a-frame show #f)`.

The initial `FRAME-WIDTH` and `FRAME-HEIGHT` values were chosen arbitrarily. As text field and button items are added to the frame, these values can be adjusted so that the frame fits around the controls.

Text field and button items are added to the frame through a panel. Roughly, a panel is a sub-window in a frame that is used to group control items. The items in this example are added to the panel in a top-left to bottom-right order. The arrangement of items within a panel is mostly automatic, but the panel method `new-line` is used to guide this arrangement.[1]

Creating an instance of `wx:panel%` adds a panel to the frame:

```
(define a-panel
  (make-object wx:panel% a-frame ; Panel is in a-frame
               0 0 FRAME-WIDTH FRAME-HEIGHT)) ; Occupy whole frame
```

The first initialization parameter for the panel specifies its graphical parent, `a-frame`. When the panel is created, it installs itself as a child of `a-frame`. The remaining parameters specify the location and size of the panel within the frame. Loading the project at this point produces a frame containing an empty panel. However, we can now add controls to the panel, starting with the top-left element, the "Name" text field:

```
(define NAME-WIDTH 300)
(define name-text
```

[1] MrEd also provides tools for constraint-based geometry management of items, panels, and frames, but the simpler line-based arrangement suffices for this example.

```
(make-object wx:text% a-panel
             (lambda (self event) (refresh-number-info))
             "Name" "" -1 -1 NAME-WIDTH -1))
```

The class `wx:text%` implements a simple text field control. The first initialization parameter is the control's graphical parent, `a-panel`. The second parameter is a "callback" procedure that is invoked whenever this control receives keyboard input. When the text in this control is changed, a new phone number search will be performed; for now, the serach procedure `refresh-number-info` remains unspecified. The third initialization parameter specifies a label for the item, and the fourth parameter is the initial string value in the control. The last four parameters specify the location and size of the control.

Loading the project now creates a frame with a text field control. The new control is graphically functional, i.e., text can be enetered into the field, but the field's value is not yet used to find a phone number. Instead, typing in the text field triggers an error, because `refresh-number-info` is not bound.

The development of the interactive phone book continues in this way for the remaining graphical elements:

```
(define number-selector
  (make-object wx:radio-box% a-panel
               (lambda (self event) (refresh-number-info))
               "" -1 -1 -1 -1  ; Default position and size
               (list "Home Number" "Office Number")))

(define quit-button
  (make-object wx:button% a-panel
               (lambda (self event) (send a-frame show #f))
               "Quit"))

(send a-panel new-line)
(define NUMBER-WIDTH 300)
(define number-text
  (make-object wx:text% a-panel
               (lambda (self event) #f) ; No event-handling
               "Number" "(Unknown)"
               -1 -1 NUMBER-WIDTH -1))
(send number-text set-editable #f)
```

At this point, all of the graphical elements have been defined. The "Quit" button is already operational; its callback procedure hides the frame.

The functionality of the rest of the interface is implmented by `refresh-number-info`:

```
(define refresh-number-info
  (lambda ()
    (let* ([name (send name-text get-value)]
           [home? (zero? (send number-selector get-selection))]
           [number-str (lookup-number name home?)]
           [number-to-show (if number-str
                               number-str
                               "(Unknown)")])
      (send number-text set-value number-to-show))))
```

This procedure extracts the current values of the `name-text` text field and `number-selector` toggle button and then calls `lookup-number` to perform the phone book search. (We assume that this procedure is already given.) The result from `lookup-number` is put into the `number-text` display text field.

3.2 Defining Classes

To develop the interactive phone book interface, we used `define` to bind graphical objects to top-level variables. This mode is convenient for interactive development and testing. However, now that the interface has been developed, we would like to encapsulate it into a class definition. Given a class that encapsulates an interactive phone book session, we can create multiple phone book frames, each with its own state.

Figure 3 shows the interactive phone book interface encapsualted in the class called `lookup-session%`. Each top-level variable in the prototype has become an instance variable in the class definition. When an instance of `lookup-session%` is created, new graphical elements are created for the instance variables in the object. The initialization procedure, defined at the end of the `lookup-session%` definition, is called after the instance variables are initialized. This procedure makes the frame visible and disables editing in the phone number display element.

3.3 Derived Classes

We can add capabilities to the phone book interface by deriving a class `lookup-session%`. For example, the derived class `lookup-counted-session%` (see Figure 4) adds a "Number of Searches Performed" counter to the interface. This new class must override the instance variable `FRAME-HEIGHT` to make the frame taller, leaving room for a counter display; it must also override the method `refresh-number-info` to increment the counter. The new `refresh-number-info` invokes the super method `refresh-number-info` to perform the actual search.

3.4 Threads

The phone book interface triggers a phone book search every time the user types a character into the "Name" text field. As soon as the text field contains enough characters to match a phone number, the number can be displayed in the "Number" field. Unfortunately, this per-character search works well only when each search is relatively fast. If the search proves too slow, the interface becomes unresponsive; the user is forced to wait for a search result each time a character is typed. We can solve this problem in MrEd by placing each search in a separate, preemptible thread. By using the `thread` procedure, the method `refresh-number-info` can start a new thread each time it is called, returning immediately to process more user events.

3.5 C Interface

MzScheme provides an interface for external "shared objects" (or "dynamically loaded libraries") implemented in C/C++. This interface allows two-way access between Scheme

```
(define lookup-session%
  (make-class '()
    (public
      [FRAME-WIDTH 400] [FRAME-HEIGHT 175]
      [NAME-WIDTH 300] [NUMBER-WIDTH 300]
      [refresh-number-info
       (lambda ()
         (let* ([name (send name-text get-value)]
                [home? (zero? (send number-selector get-selection))]
                [number-str (lookup-number name home?)]
                [number-to-show (if number-str number-str "(Unknown)")])
           (send number-text set-value number-to-show))))]
      [a-frame (make-object wx:frame%
                            '() "Phonebook" -1 -1 FRAME-WIDTH FRAME-HEIGHT)]
      [a-panel (make-object wx:panel%
                            a-frame 0 0 FRAME-WIDTH FRAME-HEIGHT)]
      [name-text (make-object wx:text% a-panel
                              (lambda (self event) (refresh-number-info))
                              "Name" "" -1 -1 NAME-WIDTH -1)]
      [quit-button (make-object wx:button% a-panel
                                (lambda (self event) (send a-frame show #f))
                                "Quit")]
      [number-selector (begin
                         (send a-panel new-line)
                         (make-object wx:radio-box% a-panel
                                      (lambda (self event) (refresh-number-info))
                                      "" -1 -1 -1 -1
                                      (list "Home Number" "Office Number")))]
      [number-text (begin
                     (send a-panel new-line)
                     (make-object wx:text%
                                  a-panel (lambda (self event) #f)
                                  "Number" "(Unknown)" -1 -1 NUMBER-WIDTH -1))])
    (lambda ()
      (send a-frame show #t)
      (send number-text set-editable #f))))
```

FIGURE 3: Class definition for a phone book session

data and external functions. A Scheme expression can invoke external functions ("foreign call-out") and external functions can manipulate and create Scheme data or invoke Scheme procedures ("foreign call-in").

MzScheme's C interface allows bottleneck code in an application to be re-implemented in a low-level language. It also lets MrEd applications use libraries that are already implemented in C/C++. For example, if the phone book search engine is provided as a C library, we can define `lookup-number` to invoke the external search function.

```
(define lookup-counted-session%
  (make-class lookup-session%
    (inherit a-panel)
    (rename [basic-refresh-number-info refresh-number-info])
    (private [search-counter 0] counter-text)
    (public
      [FRAME-HEIGHT 210]
      [COUNTER-WIDTH 250]
      [refresh-number-info ; Increment the counter and call old refresh
       (lambda ()
         (set! search-counter (add1 search-counter))
         (send counter-text set-value (number->string search-counter))
         (basic-refresh-number-info))])
    (lambda ()
      (super-init) ; Do base class initialization
      (send a-panel new-line)
      (set! counter-text (make-object wx:text% a-panel
                                      (lambda (self event) #f)
                                      "Number of Searches Started"
                                      "0" -1 -1 COUNTER-WIDTH -1))
      (send counter-text set-editable #f))))
```

FIGURE 4: Class definition for a phone book session with a counter

External code must provide special interface functions to communicate with the Scheme interpreter. The interface functions required to implement an external `lookup-number` are shown in Figure 6. MrEd provides tools for generating these interface functions automatically.

3.6 Mixing Text and Graphics

The interface of more complex MrEd application is shown in Figure 5. This application [6] is used to perform value flow analysis on Scheme programs. To show analysis information, this application displays Scheme program text along with flow arrows and value information. The user can click on expressions and arrows in the source text to show more arrows or obtain more value information. The value information for each expression is displayed in a box, which appears embedded in the program text. The boxed value information is itself text that can be arbitrarily complex.

This application relies on MrEd's primitive text-handling capabilities for displaying the program text, displaying embedded information boxes, and tracking mouse clicks on expressions in the program. The application programmer added new classes to draw arrows and handle other mouse clicks by overriding methods in the primitive classes.

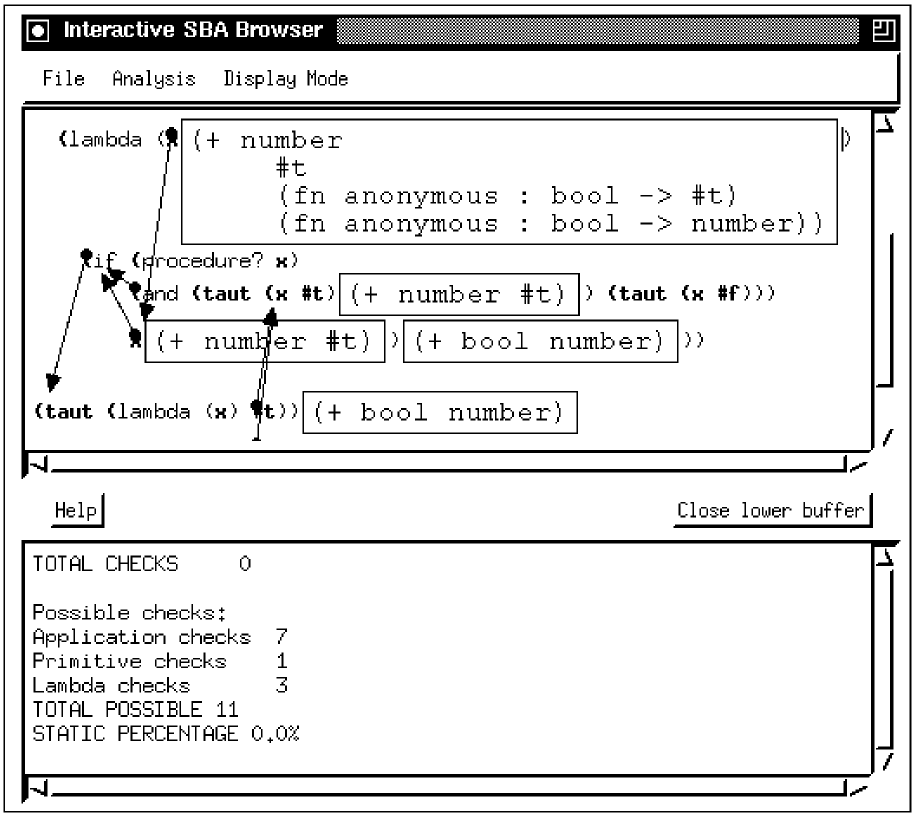

FIGURE 5: An application using MrEd's text and graphics facilities

4 Implementation

MrEd's execution kernel consists of five major components:

- The Start-up component initializes the kernel, performing platform-specific initialization and loading in the initial MrEd program. By default, the start-up component loads the development shell.
- The graphical primitives component consists of two sub-components:
  - wxWindows [14] provides the event dispatcher and portable graphical primitives. It abstracts platform-specific graphical toolboxes into a common C++ class interface.
  - wxMedia extends wxWindows with C++ classes for editing text and graphics. This component was designed specifically for MrEd.
- MzScheme is MrEd's Scheme implementation.
- The glue component integrates MzScheme with wxWindows/wxMedia.

```
/* This function is called to initialize the external module: */
void scheme_initialize(Scheme_Env* global_env) {
  /* define a new global procedure: */
  scheme_add_global("lookup-number",
                    scheme_make_prim_w_arity(do_lookup,
                                             "lookup-number",
                                             2, 2),
                    global_env);
}
/* C procedure invoked as Scheme's lookup-number: */
Scheme_Object* do_lookup(int argc, Scheme_Object* argv[]) {
  char* s;
  if (!SCHEME_STRINGP(argv[0]))
    scheme_wrong_type("lookup-number", "string", 0, argc, argv);
  /* Call the C-implemented lookup function: */
  s = lookup_number(SCHEME_STR_VAL(argv[0]), SCHEME_TRUEP(argv[1]));
  return s ? scheme_make_string(s) : scheme_false;
}
```

FIGURE 6: Interface code for C-implemented `lookup-number`

- The conservative garbage collector [2] provides memory management for the entire system.

The relationships between these components is illustrated in Figure 7. Each box in the figure represents a component; each arrow indicates that the source component depends on the destination component. The following sections describe these components in more detail.

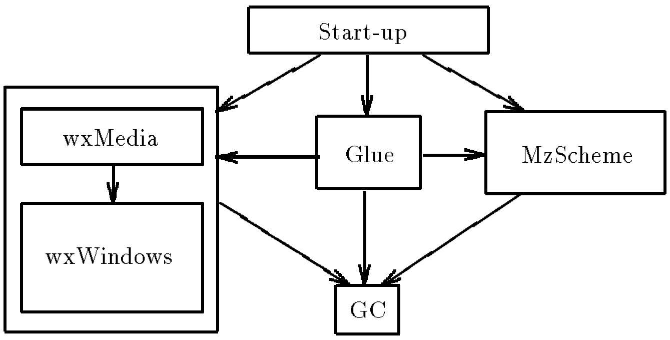

FIGURE 7: Execution kernel architecture

4.1 Editor Design

MrEd's editor primitives are built on the wxWindows canvas.[2] abstraction, which provides portable primitives for painting text and lines in a drawing context. The editor primitives in wxMedia implement two kinds of graphical elements: **editors**, for text-like manipulation of a sequential stream of editable items, and **pasteboards**, for manipulating unordered objects in a two-dimensional viewing space.

Both editors and pasteboards are divided into three functional parts:

- A **buffer** stores the state of the element and handles most events.
- A **snip** is a segment of information within the buffer. Each snip can contain a sequence of characters, a picture, or an interactive object (such as an embedded buffer).
- A **display** presents a buffer on the screen.

Two extra layers of administration have been added to manage the display-buffer and buffer-snip connections. A buffer never communicates directly with a display; instead, it always communicates with a **buffer administrator**, which relays information to the display. Similarly, a snip communicates with a **snip administrator**.

The administrative layers make the editor hierarchy flexible without adding redundant complexity. For example, a buffer can be a single item within another buffer. Without administrators, the basic buffer class would also have to contain all the functionality of a display (for the containing buffer) and a snip (for the embedded buffer). Using administrators, the same buffer class can serve as both the containing and embedded buffer and provide special administrators for the embedded buffer.

The editor primitives are fully extensible. New editor and pasteboard classes extend the user-level behavior of a buffer. New snip classes allow a buffer to deal with new kinds of graphical data. New class-specific marshalling procedures extend the file and clipboard formats by converting new kinds of data into a standard "flat" representation.

4.2 Scheme Implementation

MzScheme is a re-implementation and extension of `libscheme`[1]. MzScheme compiles Scheme code to an intermediate byte-code representation. This byte-code representation is executed by a virtual machine implemented in C. The virtual machine uses the C stack to evaluate (non-tail) procedure applications in Scheme. These applications are optimized by A-normalizing [7] the application expression and by allocating Scheme call frames on the stack.[3] Because the C stack is used for applications, first-class continuations are implemented by copying the stack into and out of the heap.

MzScheme extends standard Scheme with an object system and time-preempted threads. MzScheme's object system is loosely based on the C++ model of classes and objects, but MzScheme uses a more "Scheme-like" view of member data and procedures. Specifically, the object system does not distinguish between non-functional *instance variables* and functional

[2]The term "pasteboard" is used here in the same way that "canvas" is used in Tcl/Tk [13] A new term was selected to avoid a conflict with the wxWindows use of "canvas", which is a more primitive tool.

[3]Stack allocation is possible if the procedure does not create any closures or use `set!`.

methods; rather, a method is an instance variable that has a procedure value. Once a method is extracted from an object, it is applied like any other Scheme procedure (without passing the object as one of the arguments). MzScheme's object system is described fully in an accompanying report [8].

Threads in MzScheme are implemented as engines [11] managed by the core Scheme interpreter. When more than one thread is active, the current thread is preempted after executing a fixed number of procedure applications. A thread is suspended by grabbing its continuation and then invoking another thread's continuation. A thread can only be suspended and resumed while it is executing Scheme code.

4.3 Scheme-C++ Integration

MrEd's glue component integrates the wxMedia/wxWindows C++ classes with MzScheme. Because new Scheme classes can override methods that are `virtual` in the C++ definition, method invocation in an object can cross the Scheme-C++ boundary in either direction.

The glue functions that perform this integration are generated by a small script interpreter, `xctocc`. This interpreter takes a description of a C++ class and generates new C++ code implementing the Scheme-C++ connection. A class description in the scripting language resembles a C++ `class` declaration, but the type information is extended. Extended type information is needed because C++ data type declarations are ambiguous. For example, the C++ type `char*` can mean:

- a null-terminated string,
- a pointer to a single character,
- a pointer to a fixed-length array of characters,
- a string or the null pointer.
- a pointer to a character or the null pointer, or
- a pointer to an array or the null pointer.

For each C++ class, `xctocc` generates a new C++ class and a primitive Scheme class. Each instance of the new C++ class contains a pointer to an instance of the primitive Scheme class, and *vice-versa*.

For each method of the original C++ class, `xctocc` generates a Scheme-to-C++ glue function that is installed as a method in the primitive Scheme class. This glue function takes Scheme parameters, checks that they are the correct number and type (and, for overloaded methods, chooses a version of the method based on the number and type or arguments), and then unbundles the Scheme data into C++ data to call the original, C++-defined method.

When a method is `virtual` in the original C++ class, then the new C++ class overrides the method. When the overriding C++ method is invoked for an instance of the C++ class, it checks the associated Scheme instance for an overriding method. If a Scheme-defined method is found, then it is invoked; otherwise, the original C++ method is called.

4.3.1 Garbage Collection

The wxWindows and wxMedia libraries are written with explicit memory allocation and deallocation. The lifetimes of C++ objects in these libraries are governed by a complex set of rules. In principle, the glue component is responsible for managing the lifetime of data as it moves across the Scheme-C++ boundary. However, this management can be easily automated by using a conservative garbage collector [2]. By allocating all data, whether Scheme data or C++ data, in the conservatively collected heap, the collector can automatically manage all memory, taking into account references from both the Scheme and C++ data domains.

MrEd's versions of the wxWindows and wxMedia libraries allocate C++ objects in the conservatively collected heap. Platform-specific primitive graphical data cannot be allocated in this heap. Instead, all primitive data is encapsulated in C++ objects, explicitly managed by object constructors and destructors. Finalization callbacks, invoked by the collector, insure that the destructors is called when a C++ object is collected.

5 Comparisons to Other Systems

Tcl/Tk [13] is a currently popular GUI development package. Like MrEd, Tcl/Tk provides graphical primitives integrated with a high-level language (Tcl) and can be extended with external C/C++ code. Unlike MrEd, Tcl/Tk relies heavily on C/C++ extensions to implement large GUI applications. Tcl is slow (10 times slower than MzScheme), does not provide organizational language features (such as an object system), and does not provide preemptive threads. Tk is less widely ported than wxWindows, and does not include editing primitives comparable to wxMedia.

STk [9] provides a Scheme interface to the graphical portion of Tcl/Tk. While STk provides a better development language, including an object system, its integration with Tk is slow and unsafe.

Some packages, such as Winterp [12], Macintosh Common LISP [5], and eXene [10] connect high-level languages to platform-specific graphical primitives. Applications written with platform-specific primitives are obviously not portable. Also, the graphical primitives in these packages are typically *too* primitive.

6 Future Work

MrEd provides the correct tools for developing robust and responsive GUI applications of any size: an interactive language, an object system, preemptive threads, basic graphical primitives, and powerful editing primitives. The Programming Lanugages group at Rice has found MrEd to be an excellent development environment for implementing graphical applications that run under multiple windowing systems: DrScheme, MrSpidey, Donkey, MzCommunication. Several areas for improvement remain in MrEd.

The current integration of MzScheme with wxWindows uses a fairly direct translation of the C++ classes, methods, and functions into Scheme. As a result, the Scheme interface to the graphical primitives is biased towards an imperative style of programming. Future versions of MrEd will provide a more functional implementation of the primitive graphical classes in the future.

The wxWindows and wxMedia libraries use the standard C model of control flow: single-threaded control with simply nested function calls. In contrast, MzScheme provides full continuations and time-preempted threads. These different models of control flow are weakly integrated in MrEd at this time. Safety with continuations is maintained by prohibiting continuation invocation across calls to C++ functions. Safety with Scheme errors is provided by the glue functions, which trap Scheme errors and return default values to C++ functions. Safety with thread switching is maintained by switching threads only during the evaluation of a Scheme expressions, although the thread switch may be visible to C++ functions because the C stack is copied.

A complete environment for GUI development should provide interactive graphical tools for designing interface elements [3]. We have not yet built such tools for MrEd, but with MrEd it will be straightforward to create these tools.

References

[1] Benson Jr., Brent W. libscheme: Scheme as a C library. In *Proceedings of the USENIX Symposium on Very High Level Languages* (1994).

[2] Boehm, H., and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience* (Nov. 1988), 807–820.

[3] Cardelli, Luca. Building user interfaces by direct manipulation. In *ACM Siggraph Symposium on User Interface Software* (1988), ACM Press, pp. 152–166.

[4] Clinger, W., and Rees, J. (Eds.). The revised^4 report on the algorithmic language scheme. *ACM Lisp Pointers 4*, 3 (July 1991).

[5] Digitool, Inc. *Macintosh Common Lisp 2.0.*

[6] Flanagan, C., Flatt, M., Krishnamurthi, S., Weirich, S., and Felleisen, M. Finding bugs in the web of program invariants. In *Programming Language Design and Implementation* (1996).

[7] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation* (1993), pp. 237–247.

[8] Flatt, Matthew. The MzScheme object system. In progress.

[9] Gallesio, Erick. *STk Reference Manual, Version 2.1.6*, Feb. 1995.

[10] Ganser, E. R. and J. H. Reppy. eXene. In *Proceedings of the 1991 CMU Workshop on Standard ML* (Sept. 1991), Carnegie Mellon University.

[11] Haynes, C.T. and D.P. Friedman. Abstracting timed preemption with engines. *Journal of Computer Languages*, 2 (1987), 109–121.

[12] Mayer, N. P. *WINTERP — The OSF/Motif Widget Interpreter*, July 1994.

[13] Ousterhout, J. K. *Tcl and the Tk Toolkit*, 4 ed. Addison Wesley, Reading Massachusetts, 1994.

[14] Smart, J. *User Manual for wxWindows 1.63: a portable C++ toolkit*, Aug. 1995.