Scwm: An Extensible Constraint-Enabled Window Manager

Greg J. Badros
InfoSpace, Inc.
2801 Alaskan Way, Suite 200
Seattle, WA 98121, USA
`greg.badros@infospace.com`

Jeffrey Nichols
School of Computer Science, HCI Institute
Carnegie Mellon University, 5000 Forbes Avenue
Pittsburgh, PA 15213, USA
`jeffreyn@cs.cmu.edu`

Alan Borning
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, USA
`borning@cs.washington.edu`

ABSTRACT

We desired a platform for researching advanced window layout paradigms including the use of constraints. Typical window management systems are written entirely in C or C++, complicating extensibility and programmability. Because no existing window manager was well-suited to our goal, we developed the SCWM window manager. In SCWM, only the core window-management primitives are written in C while the rest of the package is implemented in its Guile/Scheme extension language. This architecture, first seen in Emacs, enables programming substantial new features in Scheme and provides a solid infrastructure for constraint-based window layout research and other advanced capabilities such as voice recognition. We have used SCWM to implement an interface to the Cassowary constraint solving toolkit to permit end users to declaratively specify relationships among window positions and sizes. The window manager dynamically maintains those constraints and lets users view and modify them. SCWM succeeds in providing an excellent implementation framework for our research and is practical enough that we rely on it everyday.

KEYWORDS: constraints, Cassowary toolkit, Scheme, SCWM, X/11 Window Manager

INTRODUCTION

We desired a platform for researching advanced window layout paradigms including the use of constraints. Typical window management applications for the X windows system are written entirely in a low-level systems language such as C or C++. Because the X windows libraries have a native C interface, using C is justified. However, a low-level language is far from ideal when prototyping implementations of sophisticated window manager functionality. For our purposes, a higher-level language is much more appropriate, powerful, and satisfying.

Using C to implement a highly-interactive application also complicates extensibility and customizability. To add a new feature, the user likely must write C code, recompile, relink, and restart the application before changes are finally available for testing and use. This development cycle is especially problematic for software such as a window manager that generally is expected to run for weeks at a time. Additionally, maintaining all the features that any user desires would result in terrible code bloat.

An increasingly popular solution to these problems is the use of a scripting language on top of a core system that defines new domain-specific primitives. A prime example of this architecture is Richard Stallman's GNU Emacs text editor [40]. In the twenty years since the introduction of Emacs, numerous extensible scripting languages have evolved including Tcl [34], Python [22], Perl [42], and Guile [12, 37]. Each of the first three languages was designed from scratch with scripting in mind. In contrast, Guile—the GNU Ubiquitous Intelligent Language for Extension—takes a pre-existing language, Scheme, and adapts it for use as an extension language.

We are exploring constraint-based window layout paradigms and their user interfaces. Because we are most interested in practical use of constraints, we decided to target the X windows system and build a complete window manager for X/11. We chose to use Guile/Scheme as the extension language for our project that we named SCWM—the Scheme Constraints Window Manager. The most notable feature of SCWM is constraint-based layout. Whereas typical window management systems use only direct manipulation [38] of windows, SCWM also supports a user-interface for specifying constraints among windows that it then maintains using our Cassowary Constraint solving toolkit [1]. Much of the advanced functionality of SCWM is implemented in Scheme, thus exploiting the embedded-extension-language architecture.

BACKGROUND

SCWM leverages numerous existing technologies to provide its infrastructure and support its advanced capabilities.

X Windows and `fvwm2`

A fundamental design decision for the X windows system [33] was to permit an arbitrary user-level program to manage the various application windows. This open architecture permits great flexibility in the way windows look and behave.

X window managers are complex applications. They are responsible for decorating top-level application windows (e.g., drawing labelled titlebars), permitting resizing and moving of windows, iconifying, tiling, cascading windows, and much more. Many Xlib library functions wrapping the X protocol are specific to the special needs of window managers. Because our goal is to do interesting research beyond that of modern window managers, we used an existing popular window manager, `fvwm2`, as our starting point [13]. In 1997 when the first author began the SCWM project with Maciej Stachowiak, `fvwm2` was arguably the most used window manager in the X windows community. It supports flexible configuration capabilities via a per-user `.fvwm2rc` file that is loaded once when `fvwm2` starts. To tweak parameters, end-users edit their `.fvwm2rc` files using an ordinary text editor, save the changes, then restart the window manager to activate the changes. The `fvwm2` configuration language supports a very restricted form of functional abstraction, but lacks loops and conditionals.

Despite these shortcomings, `fvwm2` provides a good amount of control over the look of windows. It also has evolved over the years to meet complex specifications (e.g., the Interclient Communication Conventions Manual [36]) and to deal with innumerable quirks of applications. By our basing SCWM on `fvwm2`, we leveraged those capabilities and ensured that SCWM was at least as well-behaved as `fvwm2`. Our fundamental change to `fvwm2` was to replace its ad-hoc configuration language with Guile/Scheme [12].

Scheme for Extensibility

Guile [12] is the GNU project's R4RS-compliant Scheme [9] system designed specifically for use as an embedded interpreter. Scheme is a very simple, elegant dialect of the long-popular Lisp programming language. It is easy to learn and provides exceptionally powerful abstraction capabilities including higher-order functions, lexically-scoped closures and a hygienic macro system. Guile extends the standard Scheme language with a module system and numerous wrappers for system libraries (e.g., POSIX file operations).

Embedded Constraint Solver

Cassowary is a constraint solving toolkit that includes support for arbitrary linear equalities and inequalities [1]. Constraints may have varying strengths, and constraint hierarchy theory [6] defines what constitutes a correct solution. We implemented the Cassowary toolkit in C++, Java, and Smalltalk, and created a wrapper of the C++ implementation for Guile/Scheme. Thus, it is straightforward to use the constraint solver in a broad range of target applications.

In addition, the Cassowary toolkit permits numerous hooks for extension. Each constraint variable has an optional attached object, and the constraint solver can be instructed to invoke a callback upon changing the value assigned to any variable and also upon completion of the re-solve phase (i.e., after all variable assignments are completed). SCWM exploits these facilities to isolate the impact of the constraint solver on existing code.

CONSTRAINTS FOR LAYOUT

Ordinary window managers permit only direct-manipulation as a means of laying out their windows. Although this technique is useful, a constraint-based approach provides a more dynamic and expressive system. In SCWM, we use the Cassowary constraint solving toolkit. On top of the primitive equation-solving capabilities of Cassowary, SCWM adds a graphical user interface that employs an object-oriented design. We specify numerous constraint classes representing kinds of constraint relationships, and instances of each class are added to the system for maintaining relationships among actual windows. The interface allows users to create constraint objects, to manage constraint instances, and to create new constraint classes from existing classes by demonstration.

Applying Constraints

Applying constraints to windows is done using a toolbar. Each constraint class in the system is represented by a button on the toolbar (figure 1). The user applies a constraint by clicking a button, then selecting the windows to be constrained. Alternatively, the user can first highlight the windows to be constrained and then click the appropriate button. Icons and tooltips with descriptive text assist the user in understanding what each constraint does. We consulted with a graphic artist on the design of our icons in an effort to make them intuitive and attractive. Preliminary user studies have demonstrated that users can determine the represented relationship reasonably well from the icons even without the supporting tooltip text.

We provide the following constraint classes in our system. Many interesting relationships are either present or can be created by combining classes in the list.

Constant Height/Width Sum Keep the total of the height/width of two windows constant.

Horizontal/Vertical Separation Keep one window always to the left of or above another.

Strict Relative Position Maintain the relative positions of two windows.

Vertical/Horizontal Maximum Size Keep height/width of a window below a threshold.

Vertical/Horizontal Minimum Size Keep height/width of a window above a threshold.

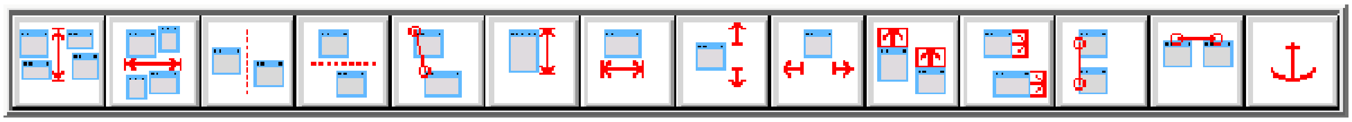

Figure 1: Our constraint toolbar. The text describes the constraint classes in the same order as they are laid out in the toolbar (from left to right).

Vertical/Horizontal Relative Size Keep the change in heights/widths of two windows constant (i.e., resize them by the same amount, together).

Vertical/Horizontal Alignment Align the edge or center of one window along a vertical/horizontal line with the edge or center of another window.

Anchor Keep a window in place.

Some of these constraint types can constrain windows in several different ways. For example, the "Vertical Alignment" constraint can align the left edge of one window with the right edge of another or the right edge of one window with the middle of another. Users specify the parameters of the relationship by using window "nonants," the ninefold analogue of quadrants (figure 2). The nonant that the user clicks in dictates the part of the window to which the constraint applies. For example, if the user selects the "Vertical Alignment" constraint and chooses the first window by clicking in any of the east nonants, and the second window by clicking on its left edge, the resulting constraint will align the right edge of the first window with the left edge of the second. This technique makes some constraint classes, such as alignment, more generally useful. It also decreases the number of buttons on the toolbar, which could otherwise become unwieldy with many narrowly-applicable constraint classes.

NW 0	N 1	NE 3
W 3	C 4	E 5
SW 6	S 7	SE 8

Figure 2: The nine nonants of a window.

Managing Constraints

Once a constraint is applied, the user still needs to be able to manage it. Users may wish to disable the constraint temporarily or remove it entirely. They may encounter an odd behavior while they are moving or resizing a window and want to discover which constraint(s) caused the unexpected result, they may simply be curious to know what constraints are applied to a given window and how that window will interact with other windows. Our constraint investigation interface allows for all of these kinds of interactions.

The constraint investigation window allows the user to enable or disable constraints using checkboxes, and to remove constraints using a delete button. The window is dynamically updated as constraints are applied and removed, and changes made in the investigator are immediately reflected in the layout of windows.

When the user moves her mouse pointer over a constraint in the investigator, the representation of that constraint is drawn directly on the windows related by the constraint (figure 3). This hint makes it easy for the user to make the correct associations between windows and constraints. Each constraint class defines its own visual representation, which in most cases closely matches the icon in the toolbar.

Enabling or disabling constraints can result in global rearrangements of windows and large changes in position. To make these discontinuities less confusing, we animate windows fluidly from their old positions and sizes to their new configuration. The animations borrow features from the Self programming environment that mimic cartoon-style animation [7].

Constraint abstractions

A problem with the interface as described thus far is that the basic constraint classes, such as "Vertical Alignment" and "Horizontal Separation," are not always sufficient to convey a user's intention fully. Our own use showed that often one needs to combine several constraints to obtain the desired behavior. A good example of this situation is tiling (figure 4), where two or more windows are aligned next to each other such that they appear to become a window unit of their own. A tiling configuration for two windows can take from three to five constraints to implement. Adding the constraints is tedious when tiling many windows, or when repeatedly tiling and untiling two windows. Certainly a "tiled windows" constraint class could be hard-coded into the system, but that just postpones the problem—some means of abstracting relationships should be provided to the end user.

Our solution to this problem is to support constraint "compositions." A composition is created using a simple programming-by-demonstration technique. We record the user applying a constraint arrangement to some windows in the workspace. The constraints used and the relationships created among the windows are saved into a new constraint class object, which then appears in the toolbar like all other constraint classes. Clicking the button in the toolbar will prompt the user to select a number of windows equal to that used in the recording. The constraints will then be

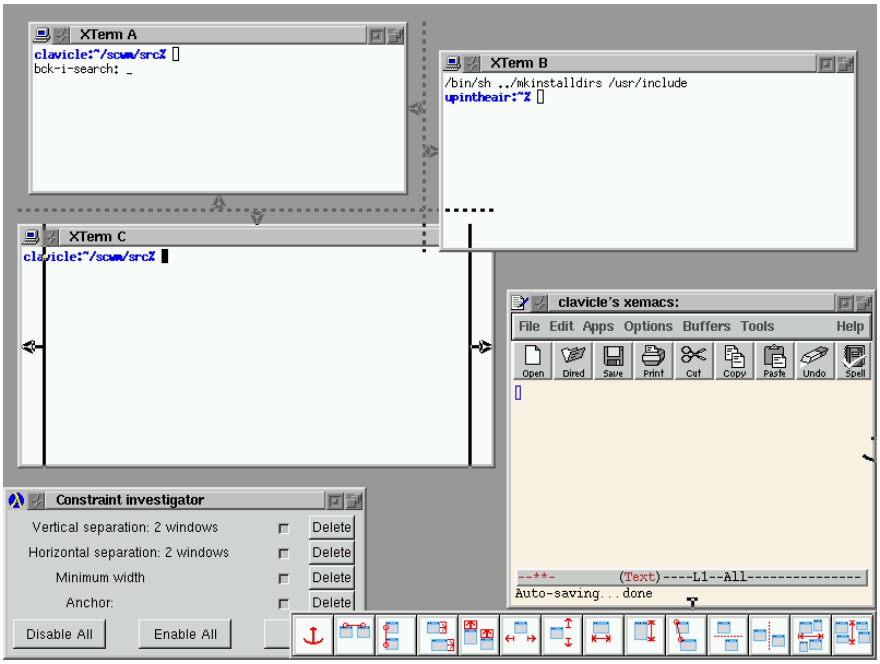

Figure 3: Visual representation of constraints. XTerm A is constrained to be to the left of XTerm B, and above XTerm C. Additionally, XTerm C is required to have a minimum width, and the XEmacs window's southeast corner is anchored at its current location. The constraint investigator that allows users to manage the constraints instances appears in the bottom left of the screen shot.

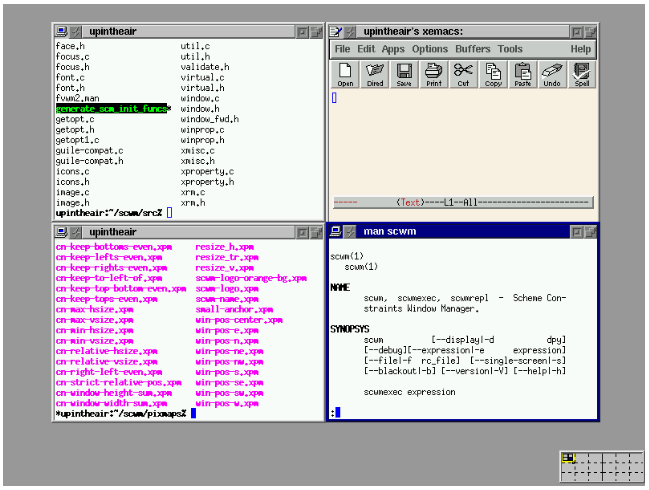

Figure 4: Four windows tiled together. Unlike tiled-only window managers, SCWM permits users to tile a subset of their windows; other windows could overlap arbitrarily.

applied in the same order as before. Compositions allow users to accumulate a collection of often-used constraint configurations that can then be easily applied.

Inferring Constraints

Our toolbar-based user interface allows flexible relationships to be specified, but many common user desires reflect very simple constraints. For example, users may place a window directly adjacent to another window and want them to stay together. Some windowing systems provide a basic "snapping" behaviour that recognizes when a user puts a window nearly exactly adjacent to another window and then adjusts the window coordinates slightly to have them snap together precisely.

In SCWM, we support a useful extension to basic snapping called "augmented snapping" [15]. Using this technique, the user has the option of transforming a snapped-to relationship to a persistent constraint that is then maintained during subsequent manipulations. When a snap is performed, instead of simply moving the window, the appropriate constraint object is created and added to the system. Such inferred constraints can be manipulated via the constraint investigator described earlier. They also can be removed by simply "ripping-apart" the windows by holding down the `Meta` modifier key while using direct manipulation to move them apart.

USABILITY STUDY

We applied a discount usability approach [32] to improve our constraint interface to managing windows.

Methodology

Six advanced computer users thought aloud while performing three tasks. Each task consists of two parts: discovery and re-creation. First, users manipulate windows with constraints already active to discover and describe those relationships (without use of the constraint investigator). After giving a correct description, they then use the interface on a second display to constrain a fresh set of windows identically. Users were given only a very minimal description of the interface.

The three constraint configurations tested were: 1) a Netscape Find dialog kept in the upper right corner of the main browser window; 2) three windows kept right-aligned along the edge of the screen such that none of the windows overlap nor leaves the top or bottom of the screen; and 3) two windows tiled horizontally.

Results

All users were able to complete their tasks. Discovering the constraints was straightforward—manipulating the windows and observing the behaviour was sufficient to deduce the relationships already present. Re-creating the configurations was more troublesome, but users still succeeded. They often used the investigator to remove incorrect constraints, but then continued onward with an alternate hypothesis.

Problems discovered

Our study uncovered numerous usability issues. The most substantial problem involved selecting window parts for the alignment constraints. When performing a vertical alignment, all that matters is whether the user clicks on the left, center, or right third of the window—it is irrelevant whether the click is in the top, middle, or bottom of the window. Our interface, however, still highlighted individual corners or edges as it does for anchor constraints where any of the nine positions is significant. Users were confused by the UI distinguishing along the irrelevant vertical dimension. We revised SCWM to highlight whole edges of windows when applying an alignment constraint.

When users began adding a constraint and wanted to cancel, they were unsure of how to abort their action. Some users clicked on the toolbar thinking that is a special window. Others discovered that clicking on the background results in an error that terminates the operation. No user realized that a right-click aborts and we now also support pressing the `Escape` key to cancel a window selection.

Other observations

The users who performed best studied the tooltip help for each of the toolbar buttons before attempting their first re-creation sub-task. We were surprised at the variety of constraints used in re-creating our configurations: no user matched the expected solution on all three tasks. In particular, the "strict relative position" constraint was used especially advantageously by users who chose to configure windows manually before applying constraints to keep the windows as they were.

```
SCWM_PROC( X_property_get,
           "X-property-get",
           2, 1, 0,
       (SCM win, SCM name, SCM consume_p))
/** Get X property NAME of window WIN. */
#define FUNC_NAME s_X_property_get
{
 SCM answer;
 VALIDARG_WIN_ROOTSYM_OR_NUM_COPY(1,win,w);
 VALIDARG_STRING_COPY(2,name,aprop);
 VALIDARG_BOOL_COPY_USE_F(3,consume_p,del);
 ...
 XGetWindowProperty(...);
 ... answer = ...;
 return answer;
}
#undef FUNC_NAME
```

Figure 5: An example SCWM primitive.

```
(define*-public (window-class
                #&optional (win (get-window)))
  "Return the class of window WIN."
  (X-property-get win "WM_CLASS"))
```

Figure 6: The "`window-class`" procedure.

Not all users discovered the constraint-visualization feature of the investigator. We now draw the visualizations whenever the user points at any part of the description, not just the enable checkbox. Also, one user wanted to modify the parameters of a constraint in the investigator window directly.

THE SYSTEM

SCWM is a complex software system that emphasizes extensibility and customizability to enable sophisticated capabilities to be developed and tested quickly and easily.

The current implementation of SCWM contains roughly 32,500 non-comment, non-blank lines of C code, 800 lines of C++ code, and 25,000 lines of Scheme code. The Guile/Scheme system is about 44,000 lines of C code and 11,500 lines of Scheme code. Finally, the Cassowary constraint solving toolkit is about 9,500 lines of C++ code in its core, plus about 1,400 lines of C++ code in the Guile wrapper. The following subsections describe various technical aspects of the implementation of SCWM in greater detail.

Basic philosophy

Our first version of SCWM was a simple derivative of its predecessor, `fvwm2`, with the ad-hoc configuration language replaced by Guile/Scheme. Like `fvwm2`, SCWM reads a startup file containing all of the commands to initialize the settings of various options. Most `fvwm2` commands have reasonably straightforward translations to SCWM sentential expressions. For example, these `fvwm2` configuration lines:

```
Style "*" ForeColor black
Style "*" BackColor grey76

HilightColor  white navyblue

AddToFunc Raise-and-Stick
+ "I" Raise
+ "I" Stick

Key s WT CSM Function Raise-and-Stick
```

are rewritten for SCWM in Guile/Scheme as:[1]

```
(window-style "*" #:fg "black"
                  #:bg "grey76")

(set-highlight-foreground! "white")
(set-highlight-background! "navyblue")

(define* (raise-and-stick
          #&optional (win (get-window)))
   (raise-window win)
   (stick win))

(bind-key '(window title) "C-S-M-s"
           raise-and-stick)
```

The simpler and more regular syntax is convenient for the end user. An even greater advantage of using a real programming language instead of a static configuration language stems from the ability to extend the set of commands (either by writing C or Scheme code) and to combine those new procedures arbitrarily.

Adding a new SCWM primitive is easily done by writing a new C function that registers itself with the Guile interpreter. For example, after using C to add the "`X-property-get`" primitive (figure 5), we can write a new procedure to report a window's class, which is just the value of its `WM_CLASS` property (figure 6). Then we can use that procedure interactively by writing:

```
(bind-key 'all "C-S-M-f"
 (lambda ()
  (let* ((win (window-with-focus))
         (class (window-class win)))
    (if (string=? class "Emacs")
        (resize-window 500 700 win)
        (resize-window 400 300 win)))))
```

The above expressions, when evaluated in SCWM's interpreter, will make the user's "`Control + Shift + Meta + f`" keystroke resize the window to either 500 × 700 pixels if the currently-focused window is an `Emacs` application window, or 400 × 300 pixels otherwise.

[1] Because the `fvwm2` configuration language is so limited, it is possible to mechanically convert to SCWM commands; we provide a reasonably-complete automated translator for this purpose.

SCWM's extensible architecture also allows Guile extensions to be accessible from the window manager. Via standard Guile modules, SCWM can read and parse web pages, download files via ftp, do regular expression matching, and much more. In fact, nearly all of the user-interface elements in SCWM are built using `guile-gtk`, a Guile wrapper of the GTk+ toolkit.

Binary Modules

Because each user only needs a subset of the full functionality that SCWM provides, it is important that users only pay for the features they require (in terms of size of the process image). Guile, unlike Emacs Lisp, allows new primitives to be defined by dynamically-loadable binary modules. Without this feature, all primitives would need to be contained in the SCWM core, thus complicating the source code and increasing the size of the resulting monolithic system.

The voice recognition module based on IBM's ViaVoice™ software illustrates the benefits of dynamically-loaded extensions. Those users who do not to use that feature—perhaps because the library is not available on their platform or perhaps because they have no audio input device—will never have the module's code loaded.

Implementing the module was also straightforward. After getting a sample program from IBM's ViaVoice™ voice recognition engine working, it required less than six hours of development effort to wrap the core functionality of the engine with a Scheme interface. A grammar describes the various utterances that SCWM understands, and the C code asynchronously invokes a Scheme procedure when a phrase is recognized. Because those action procedures are written in Scheme, the responses to phrases can be easily modified and extended without even restarting SCWM.

Graphical configuration

Another example of the extensibility that Guile provides SCWM is the `preferences` system for graphical customization. Novice SCWM users are unlikely to want to write Scheme code to configure the basic settings of their window manager, such as the background color of the currently-active window's titlebar. A graphical user interface is necessary to manage these settings, but there are potentially a huge number of configurable parameters. Undisciplined maintenance of a user interface for those options would be tedious and error-prone.

Fortunately, SCWM can leverage its Scheme extension language to ease these difficulties. The `defoption` module provides a macro `define-scwm-option` that permits declarative specification of a configuration option.[2] To expose a graphical interface to the

[2] Recent versions of Emacs [40] provide a similar feature in their "cus-

highlight-background configuration variable, the SCWM developer need simply write:

```
(define-scwm-option
  *highlight-background* "navy"
 "The bg color for focused window."
 #:type 'color
 #:group 'face
 #:setter (lambda (v)
            (set-highlight-background! v))
 #:getter (lambda () (highlight-background)))
```

This code states that *highlight-background* is an end user configurable variable that will contain a value that is a color. It also specifies that the variable can be grouped with other variables into a face category. Finally, setter and getter procedures are specified to teach SCWM how to alter and retrieve the value.

The preferences module then accumulates all of these specifications and dynamically generates the user interface shown in figure 7.[3] This modular approach also enforces the separation of the visual appearance from the desired functionality—a visually-distinct notebook-style interface with the same functionality is also available.

Connecting to Cassowary

The most important module for our research on advanced window layout paradigms is the wrapper of the Cassowary constraint solving toolkit. To connect the constraint solver with the window manager, the variables known to the solver must relate to aspects of the window layout. Each application window object contains four constrainable variables: x, y—the offsets of the window from the top-left corner of the virtual desktop); and width, height—the dimensions of the window frame in pixels. When Cassowary finds a new solution to the set of constraints, it invokes a hook for each constraint variable whose value it changes, and invokes another hook after all changes have been made. For SCWM, the constraint-variable-changed hook adds the window that embeds that constraint variable to its "dirty set," and the second hook repositions and resizes all of the windows in the dirty set.

In each window object, the constrainable variables that correspond to the window's position and size mirror the ordinary integer variables that the rest of the application uses. The hooks copy the new values assigned to the constrainable variables into the ordinary variables. This technique avoids modifying the vast majority of the code that manipulates and manages windows. (Bjorn Freeman-Benson discusses these issues in greater detail [11].)

To make it easy for developers to express constraints among windows, the constraint variables embedded in each window are available to Scheme code via the accessor primitives window-clv-{xl,xr,yt,yb,width,height}, where, for example, -xl names the x coordinate of the left side of the window and -yb abbreviates the y coordinate of the bottom of the window.[4] Thus, to keep the tops of two window objects aligned, we can use:

```
(cl-add-constraint solver
   (make-cl-constraint
      (window-clv-yt win1) =
      (window-clv-yt win2)))
```

RELATED WORK

There is considerable early work on windowing systems [16, 17, 26, 25, 27, 23]. Many of these projects addressed lower-level concerns that a contemporary X/11 window manager can ignore. An issue that does remain is tiled vs. overlapping windows. SCWM, like nearly all windowing interfaces of the 1990s, chooses overlapping windows for their generality and flexibility. However, unlike other systems, SCWM's constraint solver can permit arbitrary sets of windows to be maintained in a tiled format of a given size.

Although there are literally dozens of modern window managers in common use on the X windowing platform, only two (besides fvwm2) are especially related to SCWM. GWM, the Generic Window Manager, embeds a quirky dialect of Lisp called "WOOL" for Window Object Oriented Language [30]. It supports programmability, and some of its packages, such as directional focus changing, inspired similar modules in SCWM. Sawfish [19] is a more recent window manager with an architecture similar to GWM and SCWM. Like GWM, it embeds its own unique dialect of Lisp (called "rep"). Both embrace the extensibility language architecture and provide low level primitives, then implement other features in their extension language. However, the embedded Lisp dialects used by GWM and Sawfish both suffer from the lack of lexical closures that Scheme provides SCWM. Neither GWM nor Sawfish has any constraint capabilities, though the hooks they provide can permit procedural implementations to approximate some of the simpler constraint-based behaviours that SCWM implements.

Various other scripting languages exist. As mentioned previously, GNU Emacs and its Emacs Lisp is similar to SCWM in philosophy. The earliest popular general-purpose scripting languages is Tcl, the tool command language [34]. John Ousterhout, Tcl's author, makes a compelling case for the advantages of scripting [35]. Tcl is an incredibly simple but under-powered language that only in the most recent versions includes real data structures. Subsequent similar languages include Python [22] and Perl [42]; both are far more feature-full languages than Tcl, but all three are more commonly

tomize" package. The layout of their user-interfaces is simpler, though, as no attempt is made to create a fully graphical interface.

[3] The user interface is written in guile-gtk, a Guile wrapper of the GTk+ widget toolkit [18] that integrates seamlessly with SCWM.

[4] For each window, explicit constraints xr = x + width and yb = y + height are added automatically by SCWM.

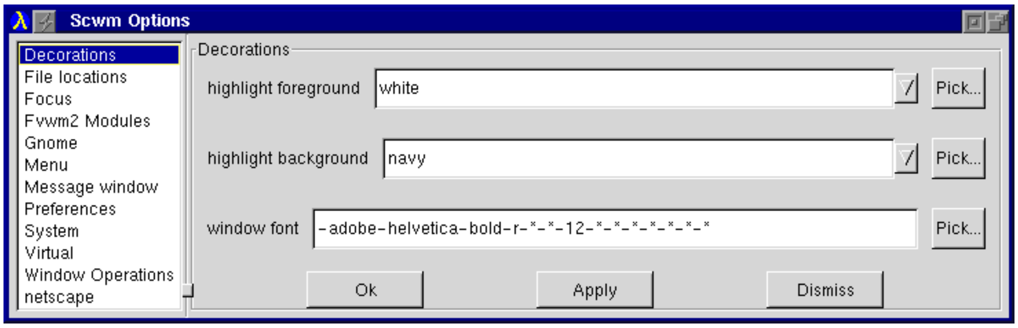

Figure 7: The automatically-generated options dialog.

used for scripting where the main control resides with the language. SCWM and Emacs both exploit their languages for embedding and invoke scripting code in response to events dispatched by C code.

There are also several other Scheme-based extension languages. Elk [10] is an early Scheme intended as an extension language but is no longer well supported. SIOD (Scheme In One Defun) [39] is an especially compact implementation of Scheme that in return compromises completeness and standards-compliance; it is embedded in the popular GIMP (GNU Image Manipulation Program) application [14] to support user-programmable transformations on images.

Numerous other application domains have used constraint solvers. Early work includes the drawing tool Sketchpad [41] and the simulation laboratory ThingLab [5]. Many other drawing programs have embedded constraint solvers over the years including Juno [31], Juno-2 [21], Unidraw [20], and Penguin [8]. Unidraw and Penguin both leverage QOCA, a constraint solver that (like Cassowary) is able to maintain arbitrary linear arithmetic constraints [24]. SCWM includes more than just constraints in its support for intelligent window layout; another paper describes some of its other layout capabilities [3].

Web browser layout presents challenges similar to window layout. Our "Constraint Cascading Style Sheets" work also embeds Cassowary and exposes a declarative specification language to web authors for describing page layout [2]. Widget layout in user interfaces is yet another two-dimensional layout problem. Amulet [29] and the earlier Garnet [28] both provided constraint solvers based on simple local propagation techniques. These solvers suffer from an inability to handle inequalities and simultaneous equations, which unfortunately arise all too often in the natural declarative specification of layout desires.

CONCLUSIONS AND FUTURE WORK

One of the most useful aspects of this research has been the continuous feedback from our end users throughout the development of SCWM. Since 1997, we have made the latest version of SCWM (along with all of its source code) available on the Internet, and have actively solicited feedback on our support mailing lists. Many of the high-level layout features were developed in response to real-world frustrations and annoyances experienced either by the authors or by our user community. Although cultivating that community has taken time and effort, we feel that the benefits from user feedback outweigh the costs.

Perhaps the most significant implementation issue for SCWM is its startup time of nearly 20 seconds on a Pentium III 450 class machine. Loading the nearly 20,000 lines of Scheme code at every restart is costly, and wasteful. To address this, we should add an Emacs-like "unexecing" capability to dump the state of a SCWM process that has all of the basic modules loaded. Although this would increase the size of the executable, it also would substantially reduce startup delays. Fortunately, after startup, SCWM's performance is indistinguishable from other window managers that are written entirely in C.

Another rich area for future work involves our constraint interface. Currently, we only support constraints among windows. It seems useful to permit the addition of "guide-line" and "guide-point" elements and allow windows to be constrained relative to them. These could, for example, be used to ensure that a window stays in the current viewport, or stays in a specific region of the display. It would also be intriguing to investigate the possibility of ghost-frame objects that are controlled exclusively by SCWM. These window frames could then hold real application windows by dragging them into the frame. This feature would permit hierarchically organizing windows, while still allowing full access to the constraint solver for non-hierarchical relationships.

We are also considering extending our voice-based interface to permit specifying constraints. In SCWM, a user can center a window simply by saying aloud "Center current window." The voice recognition interface to window layout and control encourages the user to express higher level intention: it is far more awkward to say "move window to 379, 522" than it is to say "move window next to Emacs." In this way, the voice interface usefully contrasts with direct manipulation where exact coordinates naturally result from the interaction technique. Additionally, voice-based interactions may prove especially valuable for disabled users for whom direct manipulation is difficult.

Discerning a user's true intention is an interesting complexity of the declarative specification of our current constraints interface. Consider a user who is manipulating three windows, **A**, **B**, and **C**. Suppose the user constrains **A** to be to the left of **B**, and **B** to the left of **C**. Now suppose the application displaying in window **B** terminates, thus removing that window. Should window **A** still be constrained to be to the left of window **C**? In other words, should the transitive constraint that was implicit through window **B** be preserved? The answer depends on the user's underlying desire. Providing higher-level abstractions for commonly-desired situations may alleviate this ambiguity. For example, if the user had pressed a button to keep three windows horizontally non-overlapping in a row, it is clear that window **B**'s disappearance should not remove the constraint that window **A** remain to the left of **C**.

Finally, we are especially interested in combining our work on constraints and the web [2] with this work on window layout. Web, window, and widget layout are all fundamentally related, and their similarities should ideally be factored out into a unifying framework so that advances made in any area benefit all kinds of flexible, dynamic two-dimensional layout.

ACKNOWLEDGMENTS

We thank Maciej Stachowiak, Sam Steingold, Robert Bihlmeyer, and Todd Larason for their contributions to the SCWM project. Thanks to Craig Kaplan for his helpful comments on a draft of this paper. This research has been funded in part by both a National Science Foundation Graduate Research Fellowship and the University of Washington Computer Science and Engineering Wilma Bradley fellowship for Greg Badros, and in part by NSF Grant No. IIS-9975990.

Availability

SCWM and Cassowary are both freely available on the Internet [4, 1] and are distributed under the terms of the GNU General Public License.

REFERENCES

1. Greg J. Badros and Alan Borning. The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, Seattle, Washington, June 1998. `http://www.cs.washington.edu/research/constraints/cassowary/cassowary-tr.pdf`.

2. Greg J. Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*, November 1999.

3. Greg J. Badros, Jeffrey Nichols, and Alan Borning. SCWM—an intelligent constraint-enabled window manager. In *Proceedings of the AAAI Spring Symposium on Smart Graphics*, March 2000.

4. Greg J. Badros and Maciej Stachowiak. Scwm—The Scheme Constraints Window Manager. Web page, 1999. `http://scwm.sourceforge.net/`.

5. Alan Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, March 1979. A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

6. Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992. `http://www.cs.washington.edu/research/constraints/theory/hierarchies-92.html`.

7. Bay-Wei Chang and David Ungar. Animation: From cartoons to the user interface. In *Proceedings of the 1993 ACM Conference on User Interface Software and Technology*, pages 45–55, Atlanta, Georgia, November 1993. User Interface Software and Technology.

8. Sitt Senn Chok and Kim Marriott. Automatic construction of intelligent diagram editors. In *Proceedings of UIST 1998*, San Francisco, California, November 1998.

9. William Clinger and Jonathan Rees. *Revised 4 Report on the Algorithmic Language Scheme*, November 1991.

10. Elk—the extension language kit. Web page, 1999. `http://www-rn.informatik.uni-bremen.de/software/elk`.

11. Bjorn Freeman-Benson. Converting an existing user interface to use constraints. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 207–215, Atlanta, Georgia, November 1993.

12. FSF. Guile—The GNU Ubiquitous Intelligent Language for Extension. Web page, 1999. `http://www.gnu.org/software/guile/guile.html`.

13. fvwm—the f? virtual window manager. Web page, 1999. `http://www.fvwm.org`.

14. Gimp—GNU image manipulation program. Web page, 1999. `http://www.gimp.org`.

15. Michael Gleicher. Integrating constraints and direct manipulation. In *Proceeding 1992 Symposium on Interactive 3D*, pages 171–174, 1992.

16. James Gosling. SunDew – a distributed and extensible window system. In *Methodology of Window Management*, chapter 5, pages 47–57. Springer Verlag, Heidelberg, Germany, 1986.

17. James Gosling and David Rosenthal. A window manager for bitmapped displays and unix. In *Methodology of Window Management*, chapter 13, pages 115–128. Springer Verlag, Heidelberg, Germany, 1986.

18. GTk+—the GIMP toolkit. Web page, 1999. `http://www.gtk.org`.

19. John Harper. Sawfish. Web page, 1999–2000. `http://sawmill.sourceforge.net/`.

20. Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. *An Object-Oriented Architecture for Constraint-Based Graphical Editing*, chapter 14, pages 217–238. Springer, 1995.

21. Allan Heydon and Greg Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, Digital Systems Research Center, Palo Alto, California, December 1994.

22. Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., Sebastopol, California, 1996.

23. Mark S. Manasse and Greg Nelson. *Trestle Reference Manual*. Digital Systems Research Center, December 1991. `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-068.html`.

24. Kim Marriott, Sitt Sen Chok, and Alan Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming*, 1998.

25. Brad Myers. Issues in window management design and implementation. In *Methodology of Window Management*, chapter 6, pages 59–71. Springer Verlag, Heidelberg, Germany, 1986.

26. Brad A. Myers. The user interface for Sapphire. *IEEE Computer Graphics and Applications*, 4(12):13–23, December 1984.

27. Brad A. Myers. A taxonomy of user interfaces for window managers. *IEEE Computer Graphics and Applications*, 8(5):65–84, September 1988.

28. Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, and John A. Kolojejchick. The Garnet toolkit reference manuals: Support for highly-interactive graphical user interfaces in Lisp. Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University, March 1990.

29. Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

30. Colas Nahaboo. GWM—the generic window manager. Web page, 1995. `http://www.inria.fr/koala/gwm`.

31. Greg Nelson. Juno, a constraint-based graphics system. In *Proceedings of SIGGRAPH 1985*, San Francisco, July 1985.

32. Jakob Nielson. *Usability Engineering*. Morgan Kaufmann, 1994.

33. Adrian Nye. *Xlib Programming Manual*. O'Reilly & Associates, Inc., Sebastopol, California, 1992.

34. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.

35. John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, March 1998.

36. David Rosenthal. *Inter-client Communications Convention Manual*, version 2.0 edition, 1994. `http://www.talisman.org/icccm`.

37. Peter H. Salus, editor. *Functional and Logic Programming Languages*, volume 4 of *Handbook of Programming Languages*, chapter 4. MacMillan Technical Publishin, Indianapolis, Indiana, 1998.

38. Ben Schneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

39. SIOD—scheme in one defun. Web page, 1999. `http://people.delphi.com/gjc/siod.html`.

40. Richard M. Stallman. EMACS: The extensible, customizable display editor. Technical Report 519a, Massachusetts Institute of Technology Artificial Intelligence Laboratory, March 1981. `http://www.gnu.org/software/emacs/emacs-paper.html`.

41. Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Department of Electrical Engineering, MIT, January 1963.

42. Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, California, 1996.