User Interface Specification Using an Enhanced Spreadsheet Model

Scott E. Hudson
Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

This paper describes a new interactive environment for user interface specification which is based on an enhanced spreadsheet model of computation. This environment allows sophisticated graphical user interfaces with dynamic feedback to be implemented with little or no explicit programming. Its goal is to support user interface specification by non-programming experts in human factors, visual design, or the application domain. In addition, the system is designed to allow sophisticated end-users to modify and customize their own interfaces. The system is based on a data flow model of computation. This model is presented to the interface designer in the form of a spreadsheet enhanced with new constructs for easier programming and reuse. These constructs include an improved interactive programming environment, a prototype-instance based inheritance system, support for composition, abstraction, and customization using indirect references, the addition of support for graphical inputs and outputs, and support for the encapsulation of application data structures and routines within system objects.

---

1. Introduction and Motivation

User interface software, particularly software for direct manipulation graphical user interfaces, has traditionally been very difficult to create and maintain. This has been particularly troubling since good user interface design normally relies on an iterative approach based on experience and testing with real users. A fundamental goal behind much of the research in user interface management systems (UIMS) in the last decade [Pfaf85, Myer89] has been to make it much easier to produce interfaces so that iterative development is more economically feasible. One long term goal of this work has been to allow at least partial specification of interfaces by non-programmers — particularly by non-programming experts in human factors, visual design, and the application domain. This paper describes new user interface specification techniques designed to further this goal and to extend it to the support of customization and simple modifications by end-users. Not all end-users and other non-programmers will be able to specify or even modify interfaces.

However, the work presented here is designed to extend this capability to a larger set of non-programming experts and to at least some more sophisticated end-users.

The techniques described in this paper have been implemented in a new interactive environment for user interface specification— Penguims (the Programmable ENvironment for Graphical User Interface Management and Specification). The programming environment provided by the Penguims system is based on the computational model embodied in spreadsheets. Spreadsheets are one of the few true success stories among systems for *end-user programming* — that is, systems designed to allow non-programming users to create computations of their own design. Since their introduction in the late 1970's, spreadsheets have become a standard tool for personal computing. In many cases they allow users who would be unable to construct even the simplest program in a conventional programming language to design and implement small to moderate sized custom computations.

The spreadsheet model of computation has several important properties that have lead to its success. Among these are:

- visibility of all intermediate results,

- continuous execution, and

- integration of input, output, and "program".

The techniques described in this paper are designed to provide a system for user interface specification which supports these properties and extends them in important new ways.

The basic construct upon which all spreadsheet "programs" are built is the *cell*. A spreadsheet cell consists of two parts: a value and an optional equation or formula. Within the limits of screen space, the current value of each cell is always displayed. Since intermediate results are computed in cells, they are always visible on demand. This property of visibility makes the computation more concrete and leads to a greater feeling of directness [Shne82, Shne83, Hutc86]. It also significantly decreases the difficulty of understanding and debugging a "program" since all steps can be observed directly rather than inferred from final results.

The optional equation in each cell indicates how its value may be computed or *derived* from the value of other cells. Whenever the value (or equation) of any cell in the spreadsheet changes, all cells whose values are directly or indirectly derived from that cell are recomputed and immediately updated. This provides continuous execution — no conceptually separate specification and execution phases are involved since new computations are begun whenever new inputs (or equations) arrive. This provides an exploratory environment in which experimental changes can be made (and undone) quickly and "what if" style questions can be answered with minimal effort. This also helps to break down the barrier between the static program, which is an abstraction of a class of computations, and the dynamic execution that it induces, which is more concrete. This barrier is further reduced by unifying inputs, outputs, and intermediate results (as cell values) and by placing them together with "programs" (equations) in one simple construct (the cell) that can be directly manipulated by the user. This reduction of abstraction and increase in concreteness and directness is very important since the need for a high degree of abstract thinking is one of the fundamental difficulties of programming.

Another important property of the spreadsheet model is that it can be implemented in an efficient incremental fashion. Spreadsheet "programs" represent a kind of data flow computation. This same form of computation also occurs in several other contexts. These include the attribute grammars used to control some language dependent editors and constraint systems based on one-way constraints. As a result, efficient algorithms originally developed to incrementally update systems of attributes can be used directly to update spreadsheets in a very efficient manner.

The work presented here is designed to preserve these important properties of the spreadsheet computational model while extending and enhancing them with new constructs. It retains the basic spreadsheet framework for computation, but extends this model in four general ways. It:

- provides an interactive programming environment specifically tailored for this domain,

- extends inputs and outputs from a textual interface to full graphical interfaces,

- provides new programming constructs to promote reuse, and

- provides facilities to support manipulation of external application code and data.

The Penguims interactive programming environment offers several important features to support specification of user interfaces. First, it provides structures that allow the spreadsheet computations to be organized in ways that are suitable for the interface specification domain. Conventional spreadsheets are normally structured as a fixed grid of cells. This structure is a natural framework for the row and column oriented financial calculations that spreadsheets were originally designed to support. However, in a more general context, a fixed grid makes it difficult to group cells together into logically related units. The Penguims environment drops the use of a fixed grid and introduces explicit grouping constructs: related cells are collected into *objects* and related objects may optionally be collected into *groups*. In addition, the system provides the ability to display objects and cells at different levels of detail. This allows the interface designer to make better use of limited screen space by using compact representations for most cells while increasing the screen space (and level of detail) for important or currently interesting cells. The use of multiple level displays, when combined with constructs for logically grouping related cells, allows the designer to deal with larger specifications with only a small loss in directness. Larger specifications are intrinsically more difficult to understand, consequently, the Penguims environment also provides several new features to aid in debugging.

The work presented here is also designed to go beyond the support for textual input and output found in typical spreadsheets (in the form of values entered and displayed in cells) to support a rich set of graphical I/O capabilities. As described in Section 6, graphical interactor objects (sometimes called widgets [McCo88]) from an object-oriented user interface toolkit [Henr90] may be attached to cells. These cells may take inputs from the interactor objects and/or control their graphical appearance. When combined with the computational power of the spreadsheet model, this framework provides an excellent base for supporting interfaces with a high level of dynamic feedback.

User interfaces typically must provide an interface both to a user and to an application. Consequently, the Penguims system also provides constructs to support an application interface. The system allows routines and data structures implemented in C or C++ to be linked with the system. As described in Section 8, these entities can be encapsulated inside Penguims objects so that a uniform interface can be presented to the designer and end-user. This allows the system to, among other things, support the creation of *construction kits* [Fisc88] — collections of objects that encapsulate application entities that can be "plugged together" by the end-user to form full systems or computations.

Finally, perhaps the most important new capabilities of the Penguims environment are a set of new programming constructs to promote reuse. Reuse is very important because it allows non-programmers to build on the work of programmers or more sophisticated designers. For example, one of the reasons for the success of the HyperCard system [Appl87] has been the ease with which components can be reused. A programmer or sophisticated user can create code for a complex task and attach it to a button. This code can then be reused simply by copying the button into another application. Another success with this strategy is described in more detail in [MacL90].

The Penguims system provides three specific mechanisms to promote reuse. The first provides an inheritance system. As in conventional object-oriented programming languages such as Smalltalk

[Gold83], an inheritance mechanism allows the behavior, appearance, and internal state of an object to be reused and specialized. However, to avoid the complexities often found in object-oriented type systems (see [Card85] for a survey), the work presented here does not use a conventional class-instance inheritance model. Instead, it employs a *prototype-instance* model [Lieb86, Unga87]. Under this model, new objects (*instances*) are created, not by instantiating classes, but by copying other objects (*prototypes*). This means that creating a new object which reuses all or part of a previous object can be expressed, not in terms of classes and inheritance, but in the simpler terms of copying — the user says the equivalent of "*make a new object like that one*". This approach offers a significant advantage because it makes use of the simple, easy to understand, notion of copying.

The second mechanism to promote reuse is a facility to support indirect references. As described in detail in Section 5, cells may contain special values that refer to other objects. These cells allow prototypical objects to be parameterized and composed with other objects in flexible ways. They also allow for more flexible sharing of behavior and appearance. For example, they can be used to implement a flexible style inheritance mechanism such as the one found in [Card88].

The final facility supplied by the system is a small interpreted programming language embedded in the system. While not appropriate for use by most end-users, this facility provides a "hook" for programmers or sophisticated designers to create more complex behavior without having to write C or C++ code. This facility also makes it easier to bridge the gap between event-oriented interactors, such as simulated buttons, and the data flow oriented computations of the spreadsheet. By combining this facility with prototype-instance based inheritance and shared behavior (implemented with indirect references), it becomes much easier for programmers to make their work available to end-users in a form that they can reuse. Providing this sort of smooth path between experts and novice users is crucial in the practical reuse of components.

In the next section, an overview of the architecture of the Penguims system will be presented. To set the context for the more advanced features, Section 3 will describe the basic constructs of the spreadsheet-based specification environment and Section 4 will give an example specification. Section 5 will describe programming constructs to support customization and reuse. Sections 6 and 7 will describe how graphical presentations and the interface to application entities can be handled respectively. Section 8 will consider related work, and finally, Section 9 will describe experience and present conclusions.

2. System Architecture

The Penguims system is implemented using approximately 21000 lines of C++ code and runs under the X window system [Sche86, McCo88] by means of the Artkit toolkit [Henr90].

Figure 1 illustrates the major components of the system. The user visible components of the system include: a parser for a textual input language, a graphical user interface for an interactive specification environment, and a table driven application programmer's interface for adding application specific routines and data structures to the system. In addition, internally the system maintains a set of user interface objects which are managed by a constraint evaluation system under the control of a small interpreter.

While the primary interface to the system is via the interactive environment, the system also allows user interface specifications to be entered in textual form. The textual form of specification is typically used for machine generated inputs. It is used, for example, to save the results of an interactive design session for reuse or continued execution. In addition, Penguims textual input is generated by other related tools. For example, the OPUS system provides an alternate visual programming language for specifying graphical presentations [Huds90].

Application specific routines implemented in C or C++ can be linked with the system. These are accessed by means of a set of tables as described in Section 7. Once linked with the system,
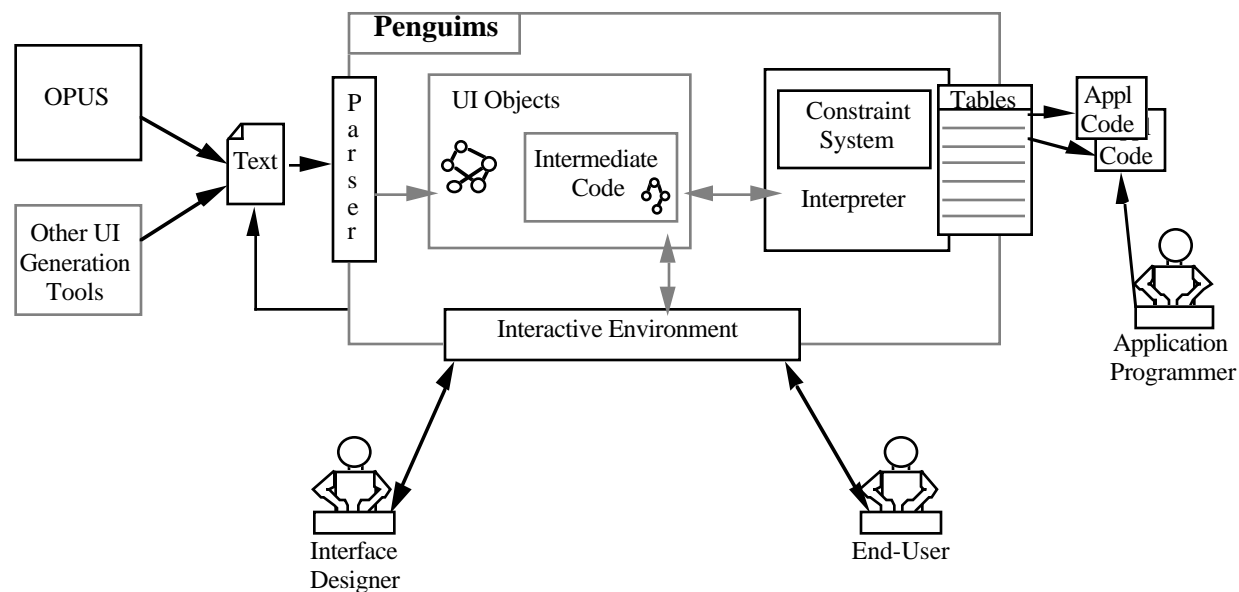
Figure 1. System Architecture

application code can be invoked from within the interpreter in the same way as built-in routines. In addition, it is possible to allow key application data structures to masquerade as Penguims objects and hence be manipulated uniformly by the interface designer. This makes it possible to create a set of application specific building blocks (e.g., a "construction kit" [Fisc88]) that make the underlying functionality of an application available in a programmable form.

The central component of the Penguims run-time architecture is an interpreter containing an incremental (one-way) constraint satisfaction system. The interpreter operates from an intermediate code produced either by the parser (from textual input) or by the interactive environment (based on user actions). This intermediate code represents expressions that may appear in spreadsheet cells as well as statements that can be executed, for example, when a button is pressed. It supports a small number of built-in data types including integers, floating point numbers, strings, bitmap images, fonts, references to objects and *code* values (as described in the Section 5). These data types can be extended with application specific data types by installing four simple routines in a table. (These routines print, copy, initialize, and dispose of a value respectively.)

Fundamental to the operation of the system is the one-way constraint satisfaction system. Any value in the system may have a defining equation attached to it. This equation describes how the value can be derived from other values in the system. Whenever a value anywhere in the system changes, all values directly or indirectly derived from it are updated automatically. The system allows displays to be controlled by cell values and arranges for the displays to be updated automatically whenever these controlling values change. Consequently, it is possible to support lexical, syntactic, and semantic feedback in a uniform framework by deriving displays directly from application and interface data.

To handle change propagation in an efficient manner, the system uses an incremental and lazy update algorithm. This algorithm was originally developed for the Higgens UIMS [Huds88] and is formally analyzed and proven correct in [Huds91]. The algorithm is incremental in that it responds to (small) changes by evaluating only the (small) set of values that might require update (rather than all attributes in the system). It is lazy in that it avoids the evaluation of attributes that

5

are not needed to produce a correct display or to invoke an application routine. This algorithm is very efficient in both an asymptotic and practical sense — performing the minimal number of attribute evaluations after any given set of changes, and using simple basic steps with very low bookkeeping storage and manipulation overhead. Although the system works on an interpreted basis and has not been extensively optimized, measured peak evaluation rates show an average update rate of approximately 6900 evaluations per second on a SPARCstation ELC. (This data comes from 20 trials of 50 reevaluations of a chain of 1000 cells connected by simple copy rules). As a result, in typical usage system performance is generally constrained more by window system drawing rates than incremental evaluation delays.

The final component of the system, the interactive designer's interface, is described in the next section. A unique feature of the Penguims system is that it uses the same interactive environment for both the interface designer and the end-user (although the interface designer has the option of hiding some objects from the end-user and disabling various editing or programming features). This has been done to promote customization by the end-user and to provide a path by which users can begin to take a direct role in development of a system. If they wish, end-users may examine parts of the spreadsheet that controls an interface to see how values change and flow through the spreadsheet. More sophisticated end-users, particularly those already familiar with spreadsheets, might then modify or customize calculations done within the interface. After more experience with the system, it may eventually be possible for an end-user to change some of the structure of the interface, add small new features, or make other significant modifications.

3. The Basic Designer's Interface

The central focus of the Penguims system is the interactive environment that forms the designer's interface. This section will consider the basic constructs of the environment in order to lay the groundwork for the more advanced features described in Sections 5 and 6. As indicated above, Penguims specifications have been structured as an enhanced form of spreadsheet. The basic
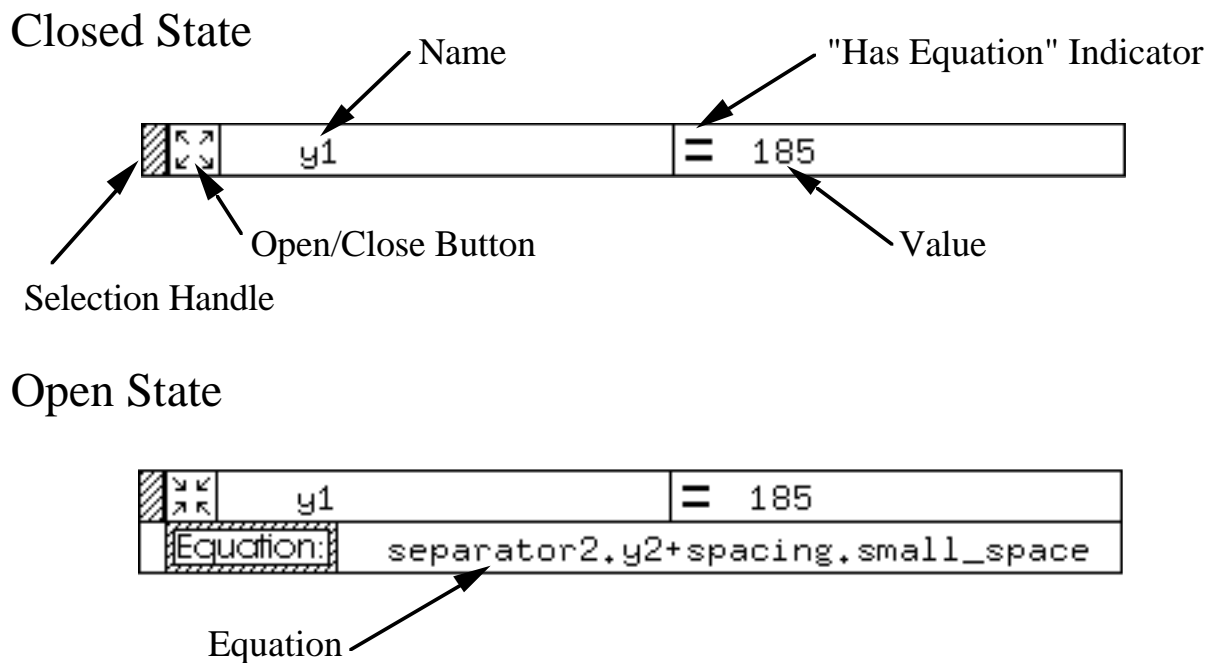

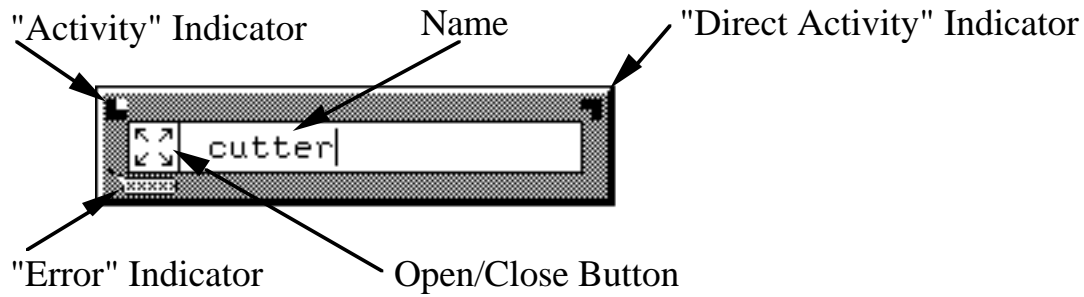Figure 2. The User Interface to a Single Cell

6

construct of the system is the *cell*. Like a normal spreadsheet cell, a Penguims cell holds a value and may optionally have a defining equation attached to it . If present, this equation determines how the value of the cell is derived from the value of other cells. Whenever values change anywhere in the system, reevaluation is done automatically to ensure that all cells directly or indirectly dependent on the changed values are consistent with their equations. Equations are expressed using conventional expression syntax with a full set of arithmetic and logical operators as well as conditional expressions, string operators, and function calls to library and application supplied routines (written in C or C++). Finally, in addition to a value and an optional equation, each cell is also given a name (in a traditional spreadsheet, cells are typically named only by their position in the fixed grid of cells).

Figure 2 shows the designer's interface to a single cell. The interface designer may modify the name, value, or equation of a cell at any time by directly editing it. After each change, the system automatically checks the modification for validity and either presents an error indication (as shown for example in Figure 3) or brings the system up to date with respect to the new value, equation, or name. Complete checking and update of the system generally occurs in less than 0.1 seconds and hence appears essentially instantaneous to the user (unfortunately, as discussed in Section 9, screen update under the X window system is not always as fast when complex graphical images are involved). Other components of the interface include an area designed to make it easier to select the cell, an open/close button, and a small icon indicating whether the cell has an equation.
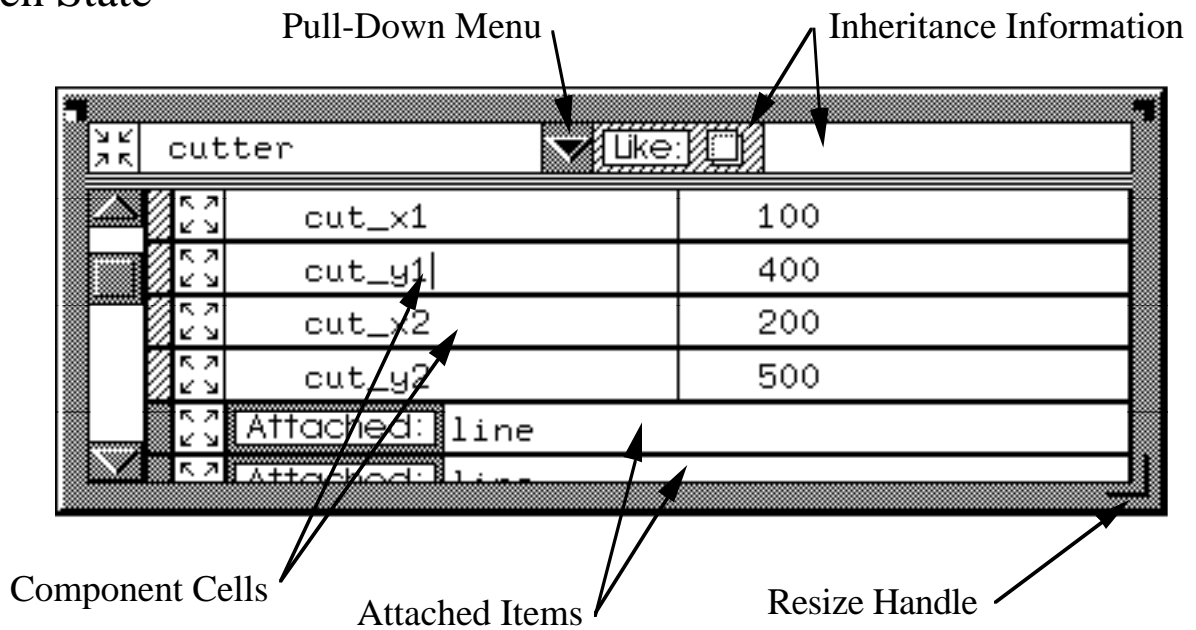
Based on experience with earlier systems (see Section 8), a central concern in the design of this interface was the efficient use of very limited screen space and the ability to support large specifications. Even a typical mid-sized interface often requires the use of more cells than can reasonably fit on the screen. This is particularly true when (parts of) the interface description must share the screen with the interface being constructed. To help with this problem, the Penguims environment uses compact displays at several levels. These displays can be changed under user control to present the level of detail needed to meet current needs. For example, the display for a cell can be presented in a compact, closed, form that hides the optional equation, or in an expanded form that allows access to all components. Similar techniques are used at each level of the interface to overcome the limitations of screen space and allow the interface designer to focus on the area(s) of current interest.

In conventional spreadsheets cells reside in a grid. Although this structure gives a predefined concrete framework suitable for some tasks and supports a form of spatial reasoning (since related cells can often be found at a given spatial offset), use of a grid makes it difficult to structure sets of cells into meaningful units (other than rows or columns). In order to facilitate the manipulation and reuse of related sets of cells, the Penguims system allows these sets of cells to be collected together into *objects*. Each Penguims object is given a name and contains a set of cells. All object names must be unique and the names of cells within a single object must also be unique. However, the same cell name may be reused across two or more objects. In programming language terms, object names are global and cell names appear within the local scope of each object. In equations, local cells may be referred to directly by name. Cells belonging to other objects can be referred to via a simple *dot* notation (e.g., obj_name.cell_name).

Closed State

"Activity" Indicator   Name   "Direct Activity" Indicator

cutter

"Error" Indicator   Open/Close Button

Open State

Pull-Down Menu   Inheritance Information

cutter   Like:

cut_x1	100
cut_y1	400
cut_x2	200
cut_y2	500
Attached:	line

Component Cells   Attached Items   Resize Handle

Error Message

Like:

1	100
1	400
2	200
2	500
line	

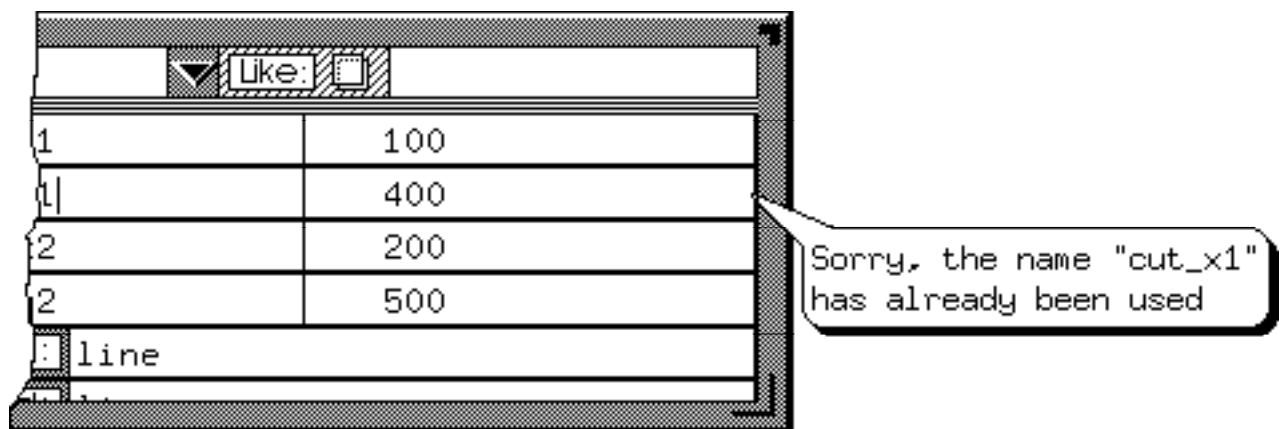Sorry, the name "cut_x1" has already been used

---

Figure 3. The User Interface to a Penguims Object

Figure 3 shows the user interface to a Penguims object. By default objects are displayed in a compact closed state as shown at the top of Figure 3. In this configuration, only the name of the object, an open/close button, two small icons (at the top left and right corners), and an optional error indicator (at the lower left corner) are visible. In this state, the object may be moved about the screen and its name may be edited, but no access is given to its cells or graphical components. To access these components, the open/close button is pressed to change the object display to its open state as illustrated in the middle of Figure 3. In the open state, the cells of the object are exposed and accessible. To maintain a compact representation, cells are presented in a scrolling list and by default are shown in a closed form with only their name and value exposed. The size devoted to the display of the cells for a particular object can be changed using the resize handle at the lower right of each expanded object.

In addition to cells, each object may also contain a set of *attached items*. These attached items form the interface to both application data structures (as discussed in Section 7) and the graphical interactor objects that make up the constructed user interface (as discussed in Section 8). For example, the object shown in Figure 3 has two line interactor objects attached to it. Each line object produces graphical output in the form of a straight line (appearing in another window), but may also accept inputs from the user by allowing its end points to be moved interactively.

Components of interactor objects (or application data structures) may be linked to cells of the object. For instance, a line object can be linked to four cells to control the coordinates of its end points. Whenever any of these cells changes value, the graphical appearance of the line will be updated automatically to reflect this change. Interactor objects can also be used as input devices. For example, a line interactor can also be used as an input device by linking it to cells in the other direction. When configured in this way, the line interactor object allows itself to be repositioned by the user and automatically updates the value of the corresponding cells. As discussed in Section 6, it is also possible for one interactor to provide both input and output.

Whenever either party in the linkage between an attached item and a cell changes, an active value mechanism (similar to the ones found in [Myer86, Schu85]) is used to notify the other party (e.g., the interface, the application, or the evaluation system). When the evaluation system is notified of a changed value, it imports the value into the system (from the interactor or application data structure) and then brings the system up to date. When an interactor is notified of a change, the redraw control portion of the system is invoked to automatically update the generated interface. This automatic update of cells and interactor objects makes it easy to provide a high degree of feedback — interactor objects can be updated dynamically based on changing cells, cells can be updated directly by the actions of interactors, and connections between these inputs and outputs can be controlled by an arbitrary set of cell equations (and/or application objects).

The compact representations for objects and cells were chosen to allow interfaces with a large number of total cells to be constructed in a manageable fashion. However, this representation has the disadvantage of being somewhat less direct — at times it hides important information. To help overcome this limitation, several features were added to the compact representation. First, whenever an error associated with one of the cells of an object occurs, the system opens the object and displays an associated message that points at the cell in question (as shown at the bottom of Figure 3). In addition, an error indicator icon is added to the lower left corner of the object. This allows any remaining errors to be detected even if the user subsequently closes the object.

In addition to an icon to report errors, the compact representation also includes two *activity indicators* (at the top left and right of the object display). These small icons spin whenever cells within the object change. The black and white indicator on the left rotates whenever any cell within the object changes value and the black and grey indicator on the right rotates whenever any cell within the object is modified directly (from the application or via the generated interface). Although difficult to show in a static picture, these dynamically moving icons are extremely effective because their motion catches the eye. They make it easy to identify cells relevant to a

9

particular interaction out of a large group of closed cells. They also serve as an effective gauge for how widespread the effects of particular changes are and how tightly interconnected various components may be.

To further improve this important debugging capability, the system provides a special *trace* operation. By pointing at any cell value or graphical object and pressing a specially designated key, the user can cause a trace message to be sent to all related cells. This message triggers the activity indicators of these objects (without changing any values) allowing them to be quickly identified and indicating any unexpected dependencies.

As a final structuring mechanism, related objects can be placed together in *groups*. The use of groups is optional — it does not effect the scope of object names or the ability to connect objects. Grouping is used primarily as a mechanism for gathering together the objects for a single application or set of related objects that can be reused as a library unit.

4. An Example Interface Specification

As an example, Figure 4 shows a group containing the objects for an interface (shown in Figure 5). This interface implements a tool to cut rectangular regions from large bitmap images (usually
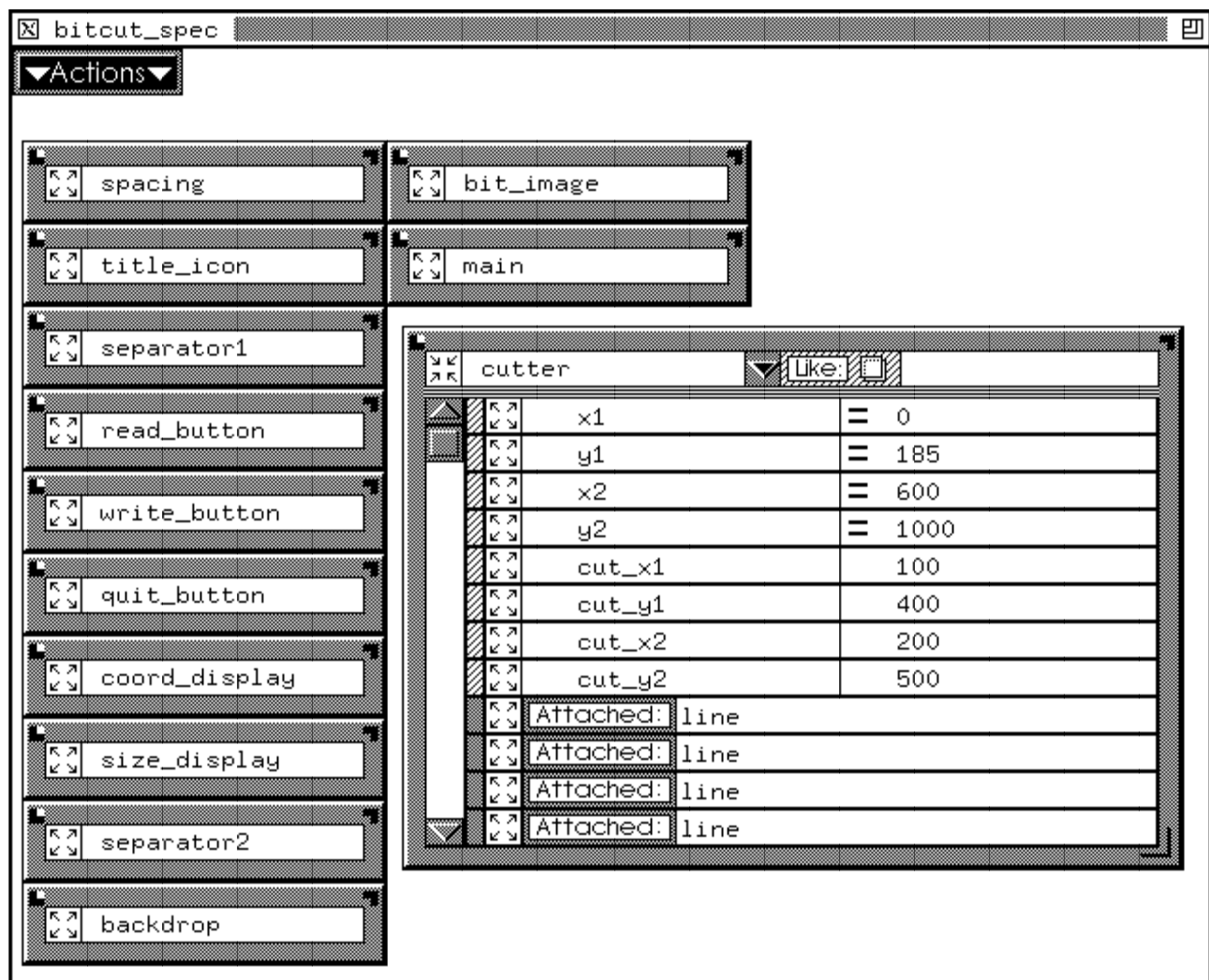

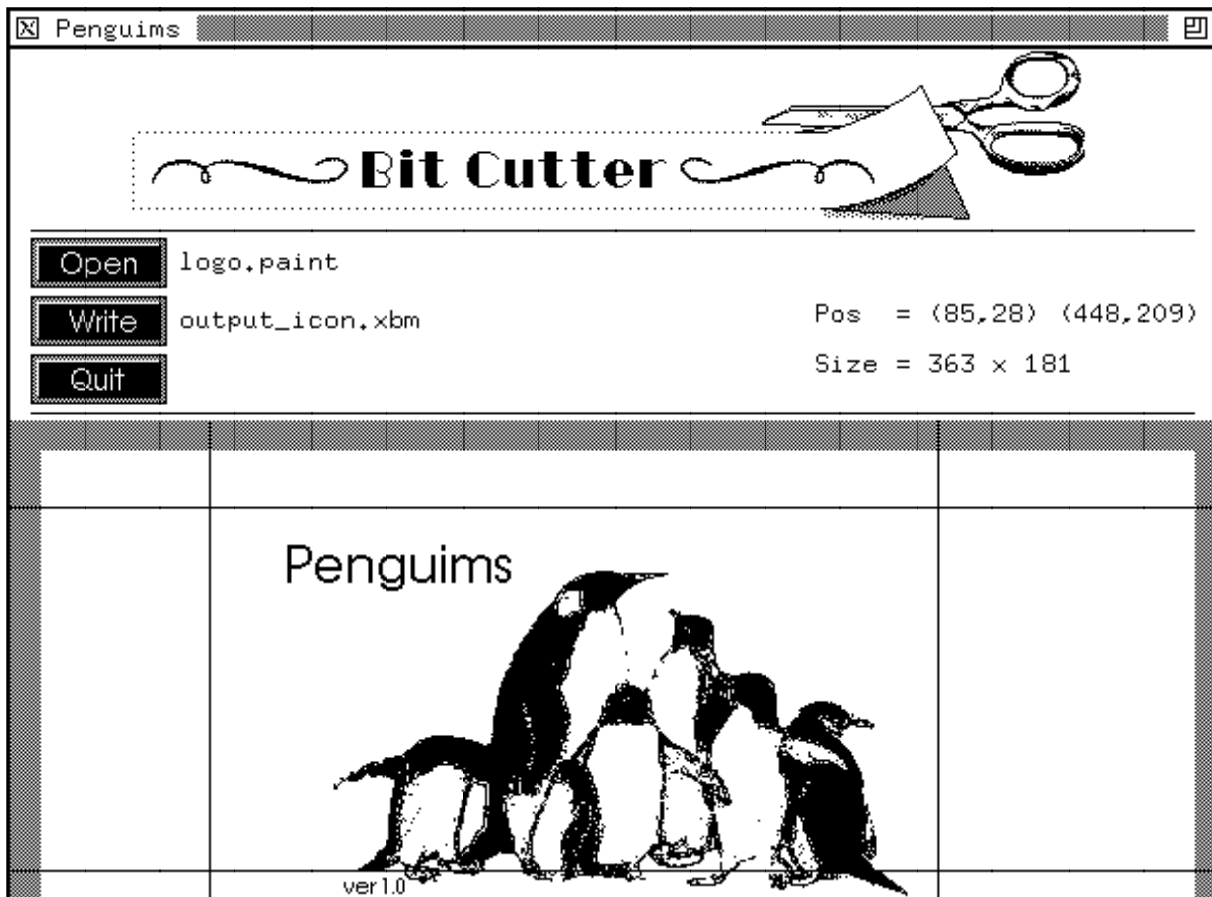Figure 4. A Group for the Sample Interface Shown in Figure 5

Figure 5. A Sample Interface Constructed With the System

constructed in a drawing program such as MacPaint™) and write them to files in X window bitmap format. It has been used in the construction of the system itself to import artwork for buttons and other interactors. As shown in Figure 4, this interface consists of 13 objects.

The spacing object is used to collect common spacing information for the other objects into a single place for easy modification . The next 8 objects create the top portion of the interface. The title_icon, separator1, and separator2 objects are responsible for the title image across the top and dialog separator lines respectively. The main dialog area of the interface is implemented with five objects. The position of each of these objects is controlled by a pair of equations. Consequently if any of object changes size (e.g., a new title icon is used) they will all automatically adjust their locations.

In the main dialog, the three buttons on the left open a new image file, write the selected area of the image to a file and quit the application, respectively. These interactors are controlled by the read_button, write_button and quit_button objects as are a pair of text editor interactors used for entering file names. The final part of the main dialog contains two text display areas at the right. These are controlled by the coord_display and size_display objects.

The final four objects in the interface are: a main object which contains special code executed at startup (in this case to resize the default graphics window that the interface appears in), a backdrop object which provides the grey background in the lower part of the interface, the

bit_image object that maintains the subject bitmap that pieces are being selected from, and finally, the cutter object.

The cutter object is the most complicated and is shown open at the bottom right of Figure 4. This object implements an interaction technique for specifying a rectangular area. This interaction technique includes four line objects that are used to define the coordinates of the rectangle. As discussed later in Section 6, these line objects are constrained to be either horizontal or vertical and are used to provide values for the cells cut_x1, cut_y1, cut_x2, and cut_y2 respectively. Note that the numbers appearing in the displays maintained by the coord_display and size_display objects are directly related to cells in the cutter object by means of simple equations. Consequently, they are automatically updated whenever parts of the cutter are moved — no special actions for screen update or information propagation are required.

5. Spreadsheet Programming Constructs For Customization and Reuse

In addition to the basic spreadsheet constructs described in the Section 3, the Penguims system introduces several new programming constructs designed to promote the reuse of interface components. An important goal of the system is to support interface specification by sophisticated end-users and other non-programmers. Strong facilities for reuse, while important to easing the interface specification task in general, are particularly important when novices or non-programmers use the system. One of the primary ways in which non-programmers (or less skilled programmers) can use the system for meaningful work is to build on the work of more sophisticated designers. Good facilities for reuse allow libraries of components to be constructed. These building blocks can then easily be composed or connected to form full interfaces. When combined with the incremental programming facilities inherent in the spreadsheet model, this provides a supportive environment for experimentation and exploratory programming. The goal is to provide a gently sloping migration path from novice to more sophisticated user of the system. Designers can begin by simply "plugging together" pre-constructed components developed by more sophisticated designers. As experience and confidence with the system increases, building blocks can first be customized in small ways, then more extensively modified. Finally, completely new components can be constructed.

To facilitate reuse, the Penguims system takes an object-oriented approach using a form of *prototype-instance* based inheritance [Lieb86, Unga87]. This form of inheritance maintains the benefits of more conventional approaches to object-oriented programming [Gold83], but does not introduce the complications of class definitions or a class hierarchy. In particular, prototype-instance based systems do not use classes. Instead any object may serve as a template or *prototype* that can be used to create new objects with the same functionality. When a new object (*instance*) is needed, it is constructed by copying (or *cloning*) it from some other object (the prototype).

This means of inheritance is particularly attractive in an interactive environment since it can be packaged in a simple and easy to use fashion. To reuse an object definition, the user can in effect ask the system to: "*make a new object like that one*". The resulting object can then either be used as is, or modified to specialize its behavior. Cloning is a very simple and easy to understand concept. However, when combined with the indirect reference facility described below, it provides most of the benefits of a full object-oriented type system.

Objects may be cloned by choosing the "Create a New Object Like This One" entry in the menu attached to the object (newly cloned objects are given system generated names which are normally changed by the user). Objects derived from prototypes are marked with the name of the prototype object in the Like field of the object. This provides a better indication of the type and purpose of the object. In addition, the system keeps track of the resulting inheritance hierarchy. When the user modifies a prototype object, the modifications are propagated to all cloned copies of the object. To control the way in which updates are propagated, inherited cells are initially marked as locked. In order to modify these inherited cells, the user must explicitly override (unlock) them.

This signals the system that the cell has been specialized in the clone and that changes to the corresponding cell in the prototype are not to be propagated to the instance.

As an example, Figure 6 shows three objects. As indicated by the Like fields, the object sub has been cloned from the object middle, which has in turn been cloned from the object super. Whenever a cell is inherited (e.g., sub.x1), it is marked with the "like" symbol at the left and its name and equation cannot be modified (as indicated by the "lock" symbol). In order to change the name or equation of an inherited cell the cell must be explicitly overridden. For example, the cell middle.x1 was originally inherited from super.x1 but then later overridden. As a result, it is not locked and its name and equation can be changed.

Whenever changes are made to cell names or equations in a prototype object, those changes are also propagated to the inherited cells of the cloned object. Figure 7, shows how various changes are propagated. Note that changing the name (or equation) of a cell in the prototype that has been overridden in the clone does not affect the clone cell or any descendants of it. So for example, changing super.x1 does not affect middle.x1 or sub.x1. However, changing middle.x1 does affect the sub.x1. Note that change propagation based on inheritance affects the name and equation of a cell, but not its value. Once the initial value of a cloned cell is copied from the prototype, both the clone and prototype values may change independently. In addition, deletion of

Figure 6. Inheritance Example

prototype objects or cells causes change propagation links to be redirected appropriately (e.g. if middle were deleted, sub.y1 would act as if it were inherited from super.y1 and sub.x1 would act as if it had been overridden).
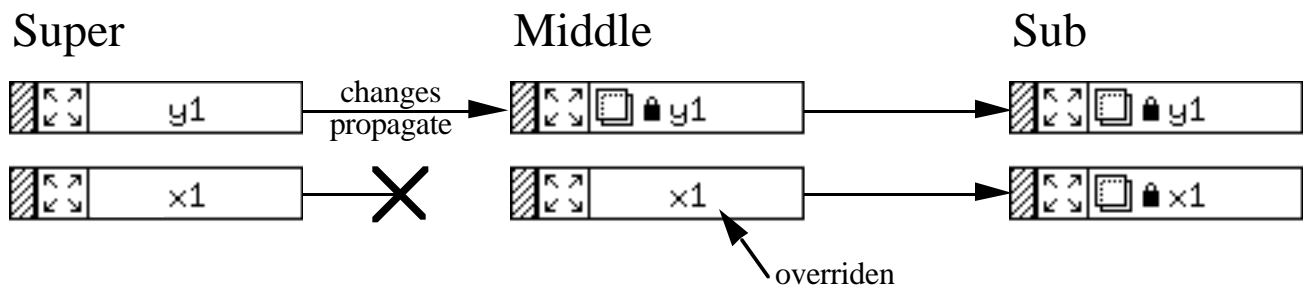
Super   Middle   Sub

y1 — changes propagate → y1 ——————→ y1

x1 —— ✕   x1 ——————→ x1

overriden

Figure 7. Propagation of Changes to Inherited Cells

boxer   Like:

style	plain
subject	dummy_obj
x1	= 8
Equation:	subject.x1 - style.min_sp
y1	= 8
Equation:	subject.y1 - style.min_sp
x2	= 98
Equation:	subject.x2 - style.min_sp
y2	= 98
Equation:	subject.y2 - style.min_sp
visible	= 1
Equation:	subject != nil
Attached:	rect

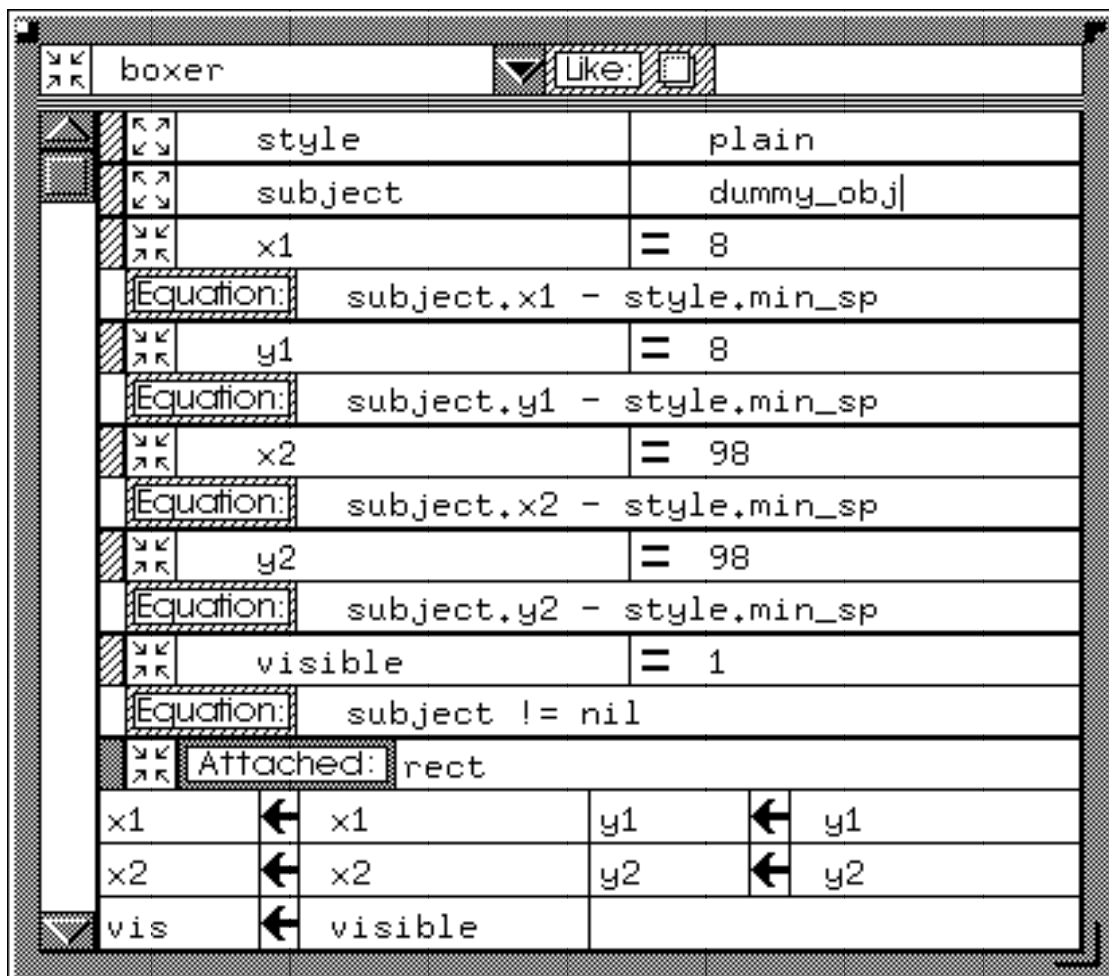| x1 | ← | x1 | y1 | ← | y1 |
| x2 | ← | x2 | y2 | ← | y2 |
| vis | ← | visible | | | |

Figure 8. An Object Which Draws a Box Around Another Object

Another important feature of the Penguims system is its support for abstraction via indirect references [Vand91]. Among the types of values that may be stored in a cell is a special *object*

pointer value (entered simply as the name of the object being referred to). These object pointer values can be used to implement shared behavior or appearance. Cells with object pointer values can be used to indirectly reference the cells of other objects — often objects designated to hold shared resources. For example, many objects contain an indirect reference cell named style which points to an object maintaining common spacings, bitmaps, and fonts for a particular style of presentation. These objects can use indirect references such as style.bold_font rather than hard coding a particular resource. This allows all the related values defining a style to be changed together and shared across objects for consistency. It also makes it possible to distinguish which aspects of an object are style independent and can be changed, and which aspects are specific to the object and must remain unchanged even when global styles are modified. This mechanism provides a style inheritance hierarchy much like that found in [Card88].

Indirect references can also be used to create objects that can be parameterized by, or composed with, other objects. For example, Figure 8 shows the specification for a prototype object, boxer, which draws a box around another object. Indirect reference is used in two places. First, the style cell provides a reference to a common style object containing spacing information. Second, the subject field holds an indirect reference to the parameter object. Thus this prototype object can be reused in adding new appearance to any object containing x1, y1, x2, and y2 cells.

Figure 9. An Extended Object Which Draws a Titled Box Around Another Object

Figure 9 shows how inheritance can be used to construct a more sophisticated titled_boxer from the boxer prototype. The titled_boxer prototype object inherits the cells and interactors of the boxer and adds title text under the box. Four new cells are added, one to hold the text string for the title, one to compute the y position of the text, another for its width, and one to hold the font used in the title. A text interactor is also added to produce the actual graphical output for the title.

15

By judicious use of indirect references it is possible to create flexible generic objects that can be placed in groups to form a kit of building blocks. When combined with the prototype-instance inheritance mechanism, this provides an excellent framework to support customization and reuse.

A final new spreadsheet programming facility offered by the Penguims system — *imperative code* — is designed primarily for use by application programmers and more sophisticated designers. In certain circumstances it is convenient to be able to use more conventional programming techniques to implement sophisticated or application dependent features. Imperative code is also useful to translate the actions of event oriented interactors, such as simulated buttons, into actions applied to spreadsheet cells. The Penguims system offers an escape mechanism to do this in the form of a small interpreted programming language. This language provides the ability to assign and use cell values within the framework of conventional control structures. In addition, it provides an interface through which application routines written in C or C++ can be called directly.

To facilitate reuse of code created by others, imperative code is represented in the system by a special form of value (a *code value*) that may be assigned to a cell and passed to interactor and application objects. Because imperative code is encapsulated in a value and may be copied or moved between cells, it is easy to implement inheritance of shared behavior. (In fact, Penguims code values are implemented in much the same way as method slots in the programming language SELF [Unga87].) As a simple example, the code value contained in a cell is copied whenever a new instance is cloned from a prototype object. In addition, code can be explicitly copied between cells. These properties make it easy to share simple objects that have attached behavior (as discussed in [MacL90]). This ability to combine and reuse the work of others is very important and is one of the reasons for the success of the HyperCard system [Appl87] as a vehicle for end-user programming.

Figure 10. An Object with Imperative Code to Support Dragging

As an example of a more sophisticated object using both imperative code and indirect references, Figure 10 shows the specification for a dragable icon prototype. This object contains an icon which is configured to be positioned by the user (as described in the next section). In addition it has imperative code that is executed when the user starts dragging the icon (contained in the **pickup** cell) and when the user stops dragging the icon (contained in the **release** cell). This code registers the object with a drag manager object (named **drag**) by setting the manager's **obj** cell. The **drag.obj** cell can in turn be used by other objects which wish to provide feedback based on the position or other attributes of the object currently being dragged. For example, a specialized

version of the dragable prototype will be used to illustrate semantic feedback in the next section. In this case, an object will use an equation (referencing the drag manager object) to detect when another object is dragged "inside" of it and will provide dynamic feedback based on the properties of that object.

6. Connecting Interactive Components

The previous three sections have considered the basic interface designer's environment and the advanced spreadsheet programming features of the Penguims system. This section will consider how spreadsheet cells are connected to components of a graphical interface to provide input, output, and feedback. As indicated above, this connection is done in terms of *attached items*. An attached item represents a parameterized external object that can be linked to one or more spreadsheet cells. These external objects can be either interactor objects (sometimes called widgets [McCo88]) or application objects (discussed in the next section).

Each attached item is of a specific object type. The set of possible object types is stored in a table linked with the Penguims system and can be easily expanded by writing a small amount of code. Each type of attached object supports a certain set of parameters. These parameters represent the "programming" interface to the object. Each parameter of the object can control the value of a cell, be controlled by the value of a cell, or both. In the case of interactor objects, this allows the interactor to provide input values to a cell, produce output based on a cell, or both.
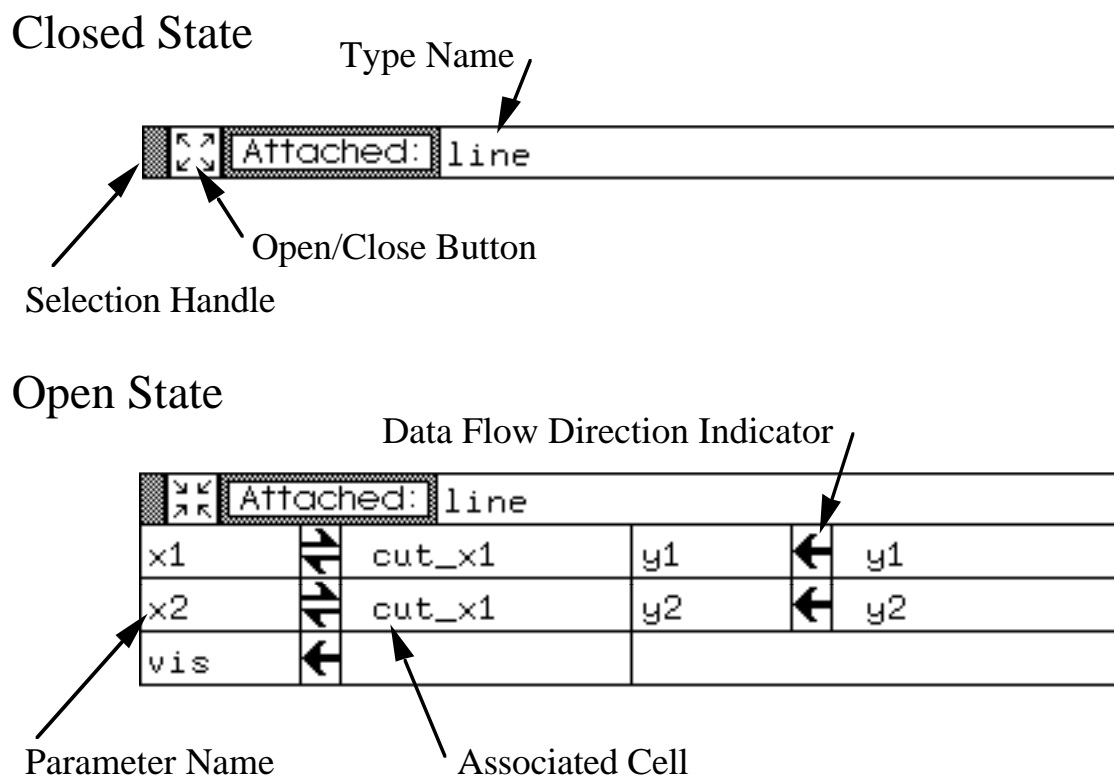

Figure 11. The Designer's Interface to a Line Interactor Object

As an example, Figure 11 shows the designers interface to a line interactor object — in this instance, the line used to control the first x value of the rectangle specified by the "cutter" object shown in Figure 4. A line interactor draws a single line on the screen and can optionally support

the repositioning of its end points by the user. A line interactor has 5 parameters. The parameters x1, y1, x2, and y2 determine the position of the end points of the line, while the vis parameter determines if the object is visible or invisible on the screen. As with other parts of the designer's interface, the interface to an attached item is by default presented in a compact form showing only its type. When expanded, the interface to each parameter becomes accessible. Each parameter is listed with its name, a small icon which can be manipulated to determine information flow direction(s), and a cell name.

This item has been set up so that the parameters y1 and y2 are controlled by the cells y1 and y2 respectively. This is indicated by the fact that the data flow direction icons point from the cell to the parameter name. As a result, whenever either of these cells change value, the y position of one end of the line will be repositioned automatically. In this example, the y1 and y2 parameters produce output based on a cell but do not accept user inputs. On the other hand, the x1 and x2 parameters are configured to both control and be controlled by the value of a cell (as indicated by the double headed arrow). This means that the line interactor will allow the user to change its x coordinates and that this will result in changes to the value of the appropriate cell. Since both coordinates are linked to the same cell either may be manipulated and the line will always remain vertical. Although not shown here, it is also possible to configure parameters to only provide inputs (e.g., control a cell). Finally, if no cell is given for a parameter, it receives a default constant value.

The line interactor object type, like most types in the standard library, supports continuous interactions — its components can respond to dragging interactions and the resulting values are continuously placed in appropriate cells. In addition, the system supports continuous reevaluation of cells and updates graphical images whenever their controlling cells change value. This provides an excellent framework for the kind of dynamic feedback needed in direct manipulation interfaces. "Input" cells can be easily linked to other cells by equations and these cells can in turn be used as "output" cells that determine graphical appearance. By including application specific functions and data structures in these equations, dynamically changing semantic feedback (that is feedback that reflects not only the low level events of the interaction, but also semantic or application specific information about the objects involved) can be supported automatically.
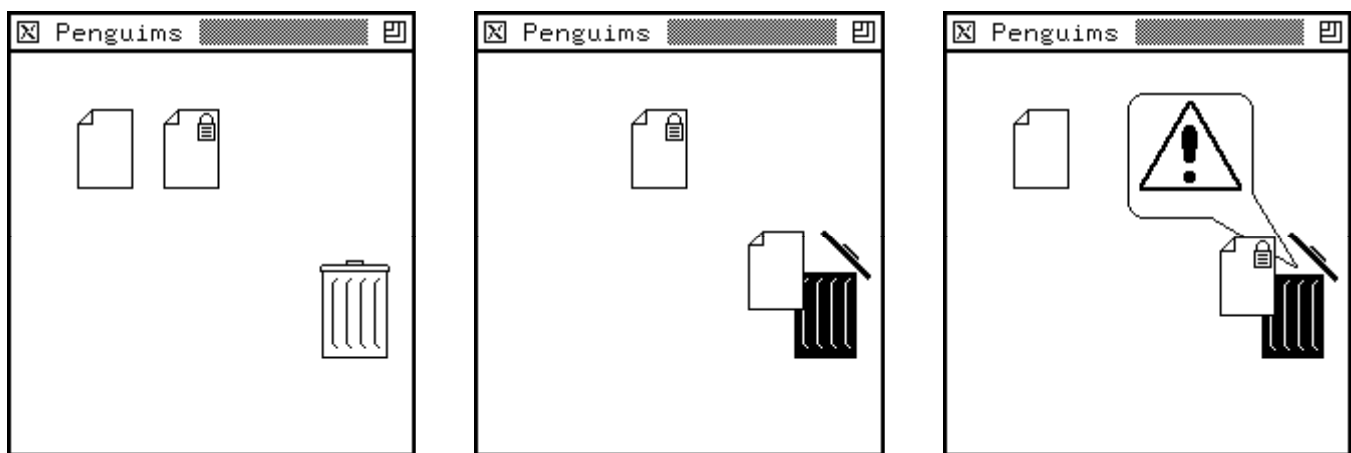
Figure 12. Semantic Feedback in a Simple "Desktop" Interface

As an example of semantic feedback, Figure 12 shows the implementation of a simplified "desktop" interface for manipulating files. A trash can is used to implement the deletion of files. Icons representing files may be dragged about the desktop by the user. When these icons are dragged over the trash can, appropriate feedback is provided based on the state of the file object — the trash can opens and if the file is locked, a warning icon is added. (Although not apparent from these static screen dumps, the feedback provided by this interface is continuous — it changes dynamically in response to all movements of the file icon).

To implement this interface, a specialized version of the dragable prototype object presented in the previous section is used. Recall that this object registers itself with a drag manager object (drag) whenever it is being dragged by the user. The trash can object uses the following equations attached to the cells inside and can_delete to "monitor" the drag.obj cell.

inside = inside_of(drag.obj.x1, drag.obj.y1, x1, y1, x2, y2)

can_delete = **not** drag.obj.locked

These equations determine 1) if the position of the currently dragged object is inside the trash can and 2) whether the currently dragged object is marked as locked. Based on the value of these two cells, the trash can object uses the following conditional expression to pick one of three bitmaps for its icon.

image = **if** inside **then** (**if** can_delete **then** open **else** warn) **else** closed

Note that the predicate used to define the can_delete cell could just as easily test any property of the object being dragged, including properties derived from more complex information stored in application data structures (as considered in the next section). Similarly, this information can be used to control the state of more complex displays. All of the tedious work, such as determining when displays should be updated, is done automatically by the system. The interface designer can concentrate on the (declarative) specifications that relate inputs and object properties to outputs. Consequently, displays with complex dynamic feedback are not significantly more difficult to construct than ordinary displays. In addition, because of the facilities for supporting reuse, similar interactions can be built using these objects as a base.

7. Supporting the Application Interface

Although the Penguims spreadsheet environment is powerful enough to build many applications in a stand alone fashion, complex, time consuming, or system dependent computations are often best implemented in a conventional general purpose programming language. To support these computations and allow them to be integrated with Penguims generated user interfaces, the system supports two forms of application interface.

First, the system allows procedures and functions written in C or C++ to be entered in a table which can be compiled and linked with the system. Entries in this table contain the name to be used for the routine, the minimum and maximum number of parameters the routine expects, and a pointer to a function that implements the routine. Once a routine has been entered in the system table, it can be called from within the equations of cells and from imperative code in the same manner as any library routine. When a call is made, the system evaluates all actual parameters and packages them in a predefined C data structure. The called routine is responsible for unpacking and type checking parameters.

As a second form of application interface, the system also allows data structures to be integrated into the system. The last section introduced how attached items can be used to link cells to interactors appearing in the user interface. Attached items can also be used for application data structures. Each field or component of the application data structure becomes a parameter for the attached item. As with interactors, each parameter of an attached item can be configured to import data from the application to the system, export data from the system to the application, or both.

19

Update and transfer of values is handled using a form of active value similar to those found in [Myer86, Schu85]. In the application code, values that flow from the application to a cell can be changed by sending a `set_value` message to the cell. For values that flow from a cell to the application, the system calls the notification routine for the item involved whenever the value in the cell is affected. In addition to the notification routine for each attached item type, the application can register a set of callback routines to be invoked after each round of evaluation. A common strategy for application update is to note the change in the notification routine but wait to respond to it until the general callback routine is invoked. This allows all the changes from a single user action to propagate to the application before being handled and thereby avoids multiple redundant updates.

Like application routines, application specific attached item types may be entered in a table, then compiled and linked with the system. Each table entry must provide the name of the attached item type, the number of parameters to the type, the names of each parameter, and a list of restrictions on input or output for each parameter. In addition, the entry must supply pointers to creation and destruction routines as well as a notification routine. The creation routine is called each time a new attached item of the given type is created. The pointer value returned by the creation routine is used in all subsequent actions involving the item (e.g., the destruction routine called when the item goes out of existence) but is otherwise uninterpreted. In most case the creation routine creates a new application data structure. However, if desired, the routine may return a pointer to an existing application object or may return some other internally meaningful value such as in index into an array.

Appropriate use of attached items makes it possible to encapsulate application data structures within Penguims objects. When combined with the proper use of prototype objects, this makes it easy to build objects that encapsulate application data structures so that they can be treated like any other object in the system.

8. Related Work

In addition to spreadsheets, which have become a central part of many personal computing environments, there are a number of other areas of research that are related to this work. The underlying computational mechanisms used by the Penguims system has roots in incremental attribute and constraint evaluation systems. Attribute grammars [Knut68, Knut71, Dera88] have long been used to describe the static semantics of programming languages. More recently, incremental attribute evaluation systems have been used as a central part of interactive programming environments (see for example [Reps85, Kais87, Aple88]). These systems offer a basic computational model which is closely related to that of spreadsheets. They use values (attributes) controlled by defining equations (attribute evaluation rules) and update these values dynamically in response to edits. One important result of this work has been the creation of very efficient algorithms for updating systems of values controlled by equations. These algorithms operate in an incremental fashion. Rather than recomputing all values after a change, they recompute only a subset of the values. An optimal incremental algorithm which recomputes the minimal subset of attributes with the minimal total overhead is described in [Reps83]. However, this algorithm is restricted to cases that can be embedded in parse trees induced by a context free grammar. The Penguims system uses a more general algorithm which is optimal in the set of attributes recomputed, but not in total overhead (see [Huds91] for full details). Other incremental algorithms for the general case are described in [Alpe87, Vand91].

Another way to view the computational model used by spreadsheets is as a form of constraint system. From this point of view, the values of cells are constrained by equations. Because the system only allows one equation to be attached to each cell and because it only propagates values from parameters of an equation to a cell and not in the other direction, it can be thought of as a *one-way* constraint system. A number of systems such as [Born81, Nels85, Born86a, Born86b, Duis86, Vand89, Hill92] have employed more general *multi-way* constraints [Lele88] in the user

interface or graphical presentation domain. Multi-way constraints are more expressive and support a larger class of computations. However, with this additional power comes some drawbacks. In particular, these techniques allow the expression of systems of constraints which may be over or under constrained. In cases where values are over or under constrained, the system can often be forced to choose between several equally viable solutions. In this situation it can become difficult to understand in advance exactly how the system will behave (one approach to overcoming this problem is described in [Born87, Free90]).

At present is unclear whether one-way or multi-way constraints offer the best solution for user interface implementation. We have chosen to use one-way constraints because they are highly predictable and understandable, can be implemented with very efficient incremental update algorithms, and because simple spreadsheets which use this model have proven in practice to be sufficiently powerful for a wide range of problems.

In addition to the general techniques of incremental attribute evaluation, and constraint evaluation, two specific systems are closely related to the work described here. First, the NoPumpG system [Lewi87, Wild90] served as the original inspiration for this work. NoPumpG was the first system that explicitly combined the spreadsheet model of computation with graphics — it allowed graphical primitives to be controlled by spreadsheet cells. This provided many of the advantages described in the introduction, particularly with respect to use by non-programmers. However, the system had several drawbacks. Most important of these were the inability to support large specifications or reuse of components. The NoPumpG system, while abandoning the fixed grid of conventional spreadsheets, did not provide an alternate structure for cells. Instead, all cells were free floating. For even mid-sized specifications, this quickly leads to problems. Available screen space fills up quickly and it becomes difficult to associate cells with the graphical elements they control. In addition, the system did not offer any specific facilities for reuse, abstraction, composition, or interface to an application.

The second system closely related to the work presented here is the C32 system [Myer91] which is part of the Garnet user interface development environment [Myer90]. This system acts as an adjunct for a separate one-way constraint system and provides a programming and debugging aid rather than a complete environment on its own. The C32 system provides a programmer's interface which allows LISP expressions specifying Garnet constraints to be entered more conveniently, but is not intended for direct use by end-users. Techniques supported by the system include the ability to enter object references by pointing at objects in a graphical display, as well as some simple demonstrational techniques for constructing constraint equations, and an inference mechanism used to support more intelligent copying of constraint equations into new contexts.

In addition to these specific systems, entirely different approaches have also been taken to the general problem of user interface specification by non-programmers. These include systems for visual specification of interfaces (see for example [Card88, Huds90]) as well as a range of *by-demonstration* techniques (see for example [Myer86, Maul89, Cyph91, Huds93, Myer93]) which allow some forms of user interface appearance and behavior to be specified without programming using inference techniques.

9. Experience and Conclusions

While it is too early to formally judge the performance of the system in the difficult realm of end-user programming (extensive user testing has not yet been performed and will be reported elsewhere), enough experience has been gained with the system to draw some positive conclusions. First, early experience with the system shows that the spreadsheet paradigm is in fact very easy to use and that the exploratory environment offered by the system makes it an excellent tool for rapidly developing user interfaces. Second, because of the automatic evaluation and screen update facilities provided by the system, it promotes a high level of dynamic feedback. In interfaces constructed by other means (e.g., window system toolkits such as Xt [McCo88]), it can be a major undertaking to create displays that move dynamically based on user input and

application semantics. In the Penguims system such interfaces are a natural result. In fact, it is so easy to create dynamic feedback that one of the problems with the system is that it is too easy to create interfaces whose display update requirements tax the limits of modern workstations (see below).

Another important lesson learned with the system involves debugging tools and support for larger specifications. The use of compact representations in the design environment was an explicit design tradeoff. It traded some of the directness resulting from the ability to see everything at once for the ability to accommodate larger designs and focus on small sections of the interface. Unfortunately, once larger specifications can be handled it also becomes easier to loose track of dependencies and relationships between cells. The debugging aids added to the system (activity indicators and the trace function) have been very important in mitigating this problem. They allow the designer to very quickly determine which objects are involved in a computation or feedback loop so that a more detailed inspection of changing values in particular cells can be made.

A final important lesson concerns the ability to embed application functions and data structures within the system in a transparent way. Although the current library of functions and prototypes is still quite small, it is already clear that much of the system's eventual usefulness will probably come from a carefully constructed library of reusable functions and objects. The ability to create Penguims objects and routines that transparently encapsulate application entities provides a powerful "hook" for expansion of its domain. For example, one of the simplest but most useful aspects of the current small library is an interface to the UNIX "system()" function call. This function allows a command line string to be passed to a shell (command interpreter) for execution. This interface, although very simple to implement, gives immediate access to a large group of existing tools. In general, with only a small amount of additional programming, it should be possible to construct a set of prototype Penguims objects to encapsulate most subroutine libraries. As a result the system provides extensibility for application programmers, but does so in a way that makes their work easily usable by non-programming designers.

In addition to positive experience with the system, certain limitations have also been uncovered. One problem with the system is that, even with a current generation workstation, redraw performance is not always adequate because it is so easy to specify dynamically changing displays. Preliminary investigations into this problem clearly indicate that actual screen update rather than the (incremental) update of cell values is to blame. Much of this problem can be traced to the overly simple redraw strategy of the toolkit used to implement the system along with the overall performance of the server-based X window system [Sche86].

In conclusion, by making use of the properties that have made spreadsheets successful end-user programming system and extending them in important new ways, the Penguims system provides an excellent environment for the rapid development of graphical user interfaces. The interactive programming environment supported by the system is easy to use and promotes the construction of interfaces with dynamic feedback — including semantic feedback — to a degree not found in other systems. In addition the specific facilities provided to support reuse (e.g., a simple prototype-instance based inheritance model and abstraction by means of indirect references) provide the opportunity to encapsulate application specific computations in a way that is much more accessible to non-programmers. As a result, it should be possible for at least some sophisticated end-users (certainly those with substantial previous spreadsheet experience) to construct or customize their own interfaces in ways that were not previously possible.

Acknowledgments

References

[Alpe87]   Alpern, B., Carle, A., Rosen, B., Sweeney, P., and Zadeck, K., "Incremental Evaluation of Attributed Graphs, IBM Research Report RC 13205, October 1987.

[Alpe88]   Alpern, B., Carle, A., Rosen, B., Sweeney, P., and Zadeck, K., "Graph Attribution as a Specification Paradigm", *Proceedings of the Symposium on Practical Software Development Environments*, Boston, November 1988, pp. 121-129.

[Appl87]   Apple Computer Inc., *HyperCard User's Guide*, Apple Computer, Inc., Cupertino, CA, 1987.

[Born81]   Borning, A., "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", *ACM Transactions on Programming Languages and Systems*, v3, (October 1981), pp. 353-387.

[Born86a]   Borning, A., "Defining Constraints Graphically", *Proceedings of CHI '86*, Boston, (April 1986), pp. 137-143.

[Born86b]   Borning, A., Duisberg, R., "Constraint-Based Tools for Building User Interfaces", *ACM Transactions on Graphics*, v5, (October 1986), pp. 345-374.

[Born87]   Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., and Woolf, M., "Constraint Hierarchies", *Proceedings of OOPSLA '87*, (October 1987), pp. 48-60.

[Card85]   Cardelli, L. and Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism" *Computing Surveys*, Dec. 1985, v18, n4,pp. 471-522.

[Card88]   Cardelli, L., "Building User Interfaces by Direct Manipulation.", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, October 1988, pp. 152-166.

[Dera88]   Deransart, P., Jourdan, M., and Lorho, B., *Attribute Grammars: Definitions, Systems and Bibliography,* Lecture Notes in Computer Science No. 323, Springer-Verlag, New York, August 1988.

[Cyph91]   Cypher, A., "EAGER: Programming Repetitive Tasks By Example", *Proceedings of CHI '91*, April 1991, New Orleans, pp. 33-40.

[Duis86]   Duisberg, R., "Animating Graphical Interfaces using Temporal Constraints", *Proceedings of CHI '86*, Boston (April 1986), pp. 131-136.

[Fisc88]   Fischer, G. and Lemke, A.C., "Construction Kits and Design Environments: Steps Towards Human Problem-Domain Communication", *Human Computer Interaction*, v3, n3, 1988, pp. 179-222.

[Free90]   Freeman-Benson, B., N., Maloney, J., and Borning A., "An Incremental Constraint Solver", *Communications of the ACM*, v33, n1, Jan. 1990, pp. 54-63.

[Gold83]   Goldberg, A., and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[Henr90]   Henry, T. R., Hudson, S. E., and Newell G. L., "Integrating Gesture and Snapping into a User Interface Toolkit", *Proceedings of the ACM Symposium on User Interface Software and Technology*, Snowbird, Utah, October 1990.

[Hill92]   Hill, R.D., "The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications", *Proceedings of SIGCHI '92*, April 1992, pp. 335-342.

[Huds88]   Hudson, S. E., and King, R., "Semantic Feedback in the Higgens UIMS", *IEEE Transactions on Software Engineering*, v14, n8, August 1988, pp. 1188-1206.

[Huds90]   Hudson, S. E. and Mohamed, S. P., "Interactive Specification of Flexible User Interface Displays", *ACM Transactions on Information Systems*, v8, n3, pp. 269-288, July 1990.

[Huds91]   Hudson, S. E., "Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update", *ACM Transactions on Programming Languages and Systems*, v13, n3, pp. 315-341, July 1991.

[Huds93]   Hudson, S. E., Hsi, C., "A Synergistic Approach to Specifying Simple Number Independent Layouts by Example", *Proceedings of InterCHI '93*, Amsterdam, April 1993, pp. 285-292.

[Hutc86]   Hutchins, E. L.,Hollan, J. D., and Norman, D. A., "Direct Manipulation Interfaces", in *User Centered Systems Design*, D. A. Norman and S. W. Draper (eds), pp. 87-124, Lawrence Erlboum Associates, Hillsdale, New Jersey, 1986.

[Kais87]   Kaiser, G. E. and Garlan, D., "MELDing Data Flow and Object-Oriented Programming", *Proceedings of OOPSLA '87*, Orlando FL, Oct 4-8, 1987, pp. 254-267.

[Knut68]   Knuth, D. E., "Semantics of Context-Free Languages", *Math. Systems Theory Journal*, v2, June 1968, pp. 127-145.

[Knut71]   Knuth, D. E., "Semantics of Context-Free Languages: Correction", *Math. Systems Theory Journal*, v5, March 1971, pp. 95-96.

[Lele88]   Leler, W., *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley, 1988.

[Lewi87]   Lewis, C. H., "NoPumpG: Creating interactive graphics with spreadsheet machinery", in *Visual Programming Environments*, Glinert E. P. (ed), IEEE Computer Society Press, Los Angeles, CA, 1990.

[Lieb86]   Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", *Proceedings of OOPSLA '86*, Portland OR, Sept. 29 - Oct. 4, 1986, pp. 214-223.

[MacL90]   MacLean, A., Carter, K., Lovstrand, L., and Moran, T., "User-Tailorable Systems: Pressing the Issues with Buttons", *Proceedings of CHI '90*, April, 1990, Seattle, WA, pp. 175-182.

[Maul89]   Maulsby, D., Kittlitz, K. A., Witten, I. H., "Metamouse: Specifying Graphical Procedures by Example", *Proceedings of CHI '89*, April 1989, Austin TX, pp. 127-136.

[McCo88]   McCormack, J. and Asente, P., "An Overview of the X Toolkit", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*,Banff, Alberta, October 1988, pp. 46-55.

[Myer86]   Myers, B. A., and Buxton, W., "Creating Highly Interactive Graphical User Interfaces by Demonstration", *Computer Graphics* , v20, n3, August 1986, pp. 249-258.

[Myer89]   Myers, B. A., "User Interface Tools: Introduction and Survey", *IEEE Software*, January 1989, pp. 15-23.

[Myer90]   Myers, B.A., Giuse, D.A., Dannenberg, R.B., Vander Zanden, B., Kosbie, D.S., Pervin, E., Mickish, A., and Marchal, P., "Comprehensive support for graphical, highly-interactive user interfaces: the Garnet user interface development environment", *IEEE Computer*, **23**(11), pp. 71-85, November 1990.

[Myer91]   Myers, B. A. ,"Graphical Techniques in a Spreadsheet for Specifying User Interfaces", *Proceedings of CHI '91*, New Orleans, May 1991, pp. 243-249.

[Myer93]   Myers, B. A., McDaniel, R. G., and Kosbie, D. S., "Marquise: Creating Complete User Interfaces by Demonstration", *Proceedings of InterCHI '93*, Amsterdam, April 1993, pp. 293-300.

[Nels85]   Nelson, G., "Juno: a Constraint-Based Graphics System", Proceedings of SIGGRAPH '85, San Francisco, CA July 1985, pp. 235-243.

[Pfaf85]   Pfaff, G., E., *User Interface Management Systems*, Springer-Verlag, New York, 1985.

[Reps83]   Reps, T., Teitelbaum, T., and Demers, A., "Incremental Context-Dependent Analysis for Language-Based Editors", *ACM Transactions on Programming Languages and Systems*, v5, July 1983, pp., 449-477.

[Reps85]   Reps, T., and Teitelbaum, T., "The Synthesizer Generator", *Proceedings of Symposium on Practical Software Development Environments*, pp. 42-48, Pittsburgh, 1985.

[Sche86]   Scheifler, R. W. and Gettys J., "The X Window System", *ACM Transactions on Graphics*, v5, April 1986, pp. 79-109.

[Schu85]   Schulert, A. J., Rogers, G. T., and Hamilton J. , "ADM - A Dialog Manager", *Proceedings CHI '85*, San Francisco, CA, April 1985, pp. 177-183.

[Shne82]   Shneiderman B., "The Future of Interactive Systems and the Emergence of Direct Manipulation", *Behaviour and Information Technology*, v1, 1982, pp. 237-256.

[Shne83]   Shneiderman B., "Direct Manipulation: A Step Beyond Programming Languages", *Computer*, v16, August 1983, pp. 57-69.

[Unga87]   Ungar, D. and Smith R. B., "Self: The Power of Simplicity", *Proceedings of OOPSLA '87*, Orlando FL, Oct 4-8, 1987, pp. 227-242.

[Vand89]   Vander Zanden, B., "Constraint Grammars — A New Model for Specifying Graphical Applications", *Proceedings of CHI '89*, Austin, TX, April 1989, pp. 325-330.

[Vand91]   Vander Zanden, B., Myers, B. A., Dario, G., Szekely, P., "The Importance of Pointer Variables in Constraint Models", *Proceedings of the ACM Symposium on User Interface Software and Technology*, November 1991.

[Wild90]   Wilde, N, and Lewis, C. H., Spreadsheet-based Interactive Graphics: From Prototype to Tool, *Proceedings of CHI'90*, Seattle, WA, April 1990, pp. 153-159.