

**SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND**

**Queensland 4072
Australia**

TECHNICAL REPORT

No. 94-9

Cover Your Self With Skin

John Hosking, Stephen Fenwick, Rick Mugridge and John Grundy

March 1994

Phone: +61 7 365 1003

Fax: +61 7 365 1533

Cover Your Self With Skin

John Hosking, Stephen Fenwick, Rick Mugridge

Department of Computer Science
University of Auckland
Auckland, New Zealand
j.hosking@cs.auckland.ac.nz

John Grundy

Department of Computer Science
The University of Waikato
Hamilton, New Zealand
j.grundy@cs.waikato.ac.nz

A visual functional language for constructing user interface components is described. The language, *Skin*, assumes a simple object-oriented interface to the underlying application and components may flexibly adapt to changes in the application. The language avoids the need for absolute or relative coordinate specification for subcomponents. An interesting feature of the language is that meaningful icons for user-defined functions are able to be automatically constructed using prototype applications of the function.

1. Introduction

The definition of user interface components can often be a tedious process, involving trial and error construction of components using a textual definition language [12]. More rapid construction is provided by using direct manipulation user construction tools [10,3], which provide immediate feedback on the appearance of components. However, these often lead to somewhat inflexible components due to the poor expressive power of the tools and notations used.

In our work, we have been particularly interested in *flexible* user interface components, i.e. those which can visualise a variety of underlying program elements and can adapt to changes in those elements (often resulting in large changes to the interface components). Fig. 1 shows some examples of such user interface components.

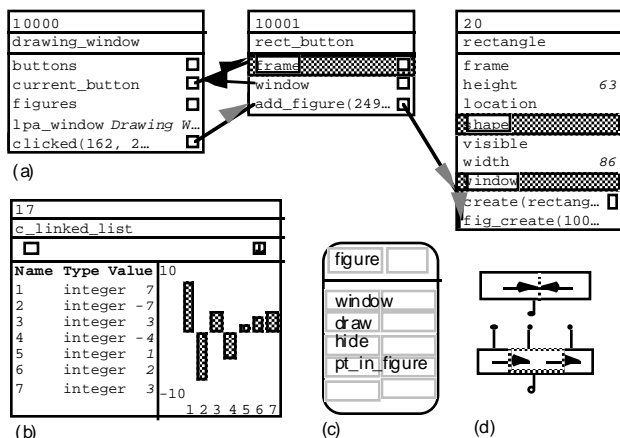


Fig. 1 Examples of flexible user interface components

Figs 1(a) and 1(b), constructed by the Cerno program debugger/visualiser [6], show components visualising the execution state of objects (or abstractions of objects). In Fig 1(a) the states of three objects are shown, with information on object attributes (highlighting => values incompatible with declared types) active methods, references to other objects (boxes plus dark arrows), and method calls (grey arrows). Fig 1(b) shows another Cerno component providing two abstract views of a linked list of objects - as a list of element number, node type and value triples, and as a bar graph. As objects change state, these user interface components must update themselves to reflect the new state. Also, regions of the components must be sensitive to user interaction. For example, clicking in the square box of an object reference attribute causes that reference to be expanded as a new interface component.

Fig. 1(c) shows an icon representing a class taken from a visual programming view of SPE, a visual/textual programming environment for object oriented programming [7]. The icon contains information about the class name and class features, which must change as the program is modified. Regions of the icon, indicated by hashed outline (not visible on the rendered icon), are sensitive to mouse clicks. Relationships between classes are indicated by connector lines between connection points on the class icons involved.

Fig. 1(d) shows two icons from the visual language described in this paper. The icons have input and output pins. These are sensitive to user interaction, acting as start and end points for "wires" connecting icons together. Some of the components in the language have an expandable number of pins: extra pins are created as existing pins are wired to.

Many user interface toolkits, such as Fabrik [10] and Interviews [11], provide reasonably good support for direct manipulation construction of traditional "dialogue box" style interface components. These are generally focused on the widget (icon composition) level, however, and construction of the sorts of flexible iconic components shown in Fig. 1 is beyond the means of most UI toolkits. These more complex icons, and editors to manipulate them, can be defined using programming techniques. These use textual, procedural languages, however, and the resulting icon definitions are often

inflexible, difficult to maintain and hard to understand due to their low-level, complex details [9].

A challenge is to be able to create succinct, generic component specifications that can be applied to visualise different underlying program entities. For example, a single specification should suffice to generate each of the three components shown in Fig 1(a), another for all class icons similar to Fig 1(c), and a third for all visual programming icons like those in Fig 1(d). The aim of the work described in this paper is to develop a visual user interface construction language. This combines the immediate feedback of direct manipulation "constructor kits" with enough expressive power to permit definition of genuinely flexible interface components, as well as more conventional dialogue box, menu, and window components.

In the next section, we review existing work in this area. Section 3 provides a brief overview of Skin, our user interface component description language, with the Skin environment described in Section 4. A more detailed description of Skin is given in Section 5. A brief description of implementation issues in Section 6 is followed by conclusions and future work in Section 7.

2. Related work

Traditional UI construction can be done via toolkits, frameworks and UIMs [11]. Interviews [11] provides a framework of object-oriented classes which are specialised to define user interface components such as edit boxes, text captions and buttons. Such systems rely on textual programming of the interface components, including absolute co-ordinates and size, and their programs must be run to test the interface appearance and functionality.

User interface builders, like Peridot [13] and FormsVBT [2], allow interface components to be specified and laid out via direct manipulation and the semantics of the interface to be programmed textually. FormsVBT keeps the graphical appearance of the interface consistent with a corresponding textual description, allowing interface components to be specified in whichever form is most convenient. A problem with this approach is the cluttering of the direct manipulation view for complex interfaces [9]. FormsVBT also provides a third view which shows the resultant user interface [3], and this can be interacted with to test the viability of the interface.

Prograph [4] provides a graphical interface builder and allows the interface semantics to be programmed visually. This method of associating user interface components and their application components is often clearer than in user interface builders with generated, textual interfaces. Fabrik [10] uses a similar dataflow

metaphor to Prograph but its data entities directly correspond to user interface components such as sliders and edit boxes. Fabrik diagrams are "always running" in that the resultant user interface is always available for testing (even when incomplete), similar to FormsVBT. This interface can be interacted with whenever desired to determine the viability of the user interface under construction. A disadvantage is the use of absolute co-ordinates for positioning interface components, making the system less adaptive to application object change.

EDGE [14] allows new graphical editors for graph-based structures to be generated from a textual specification of the graph. The graph node and edge icons, however, must be programmed to achieve complex iconic representations and behaviours. Trip 3 [12] allows new graphical editors for graph structures to be specified via visual examples of application data linked by a declarative mapping. Complex graph nodes and their arcs, however, are difficult to specify abstractly in this approach.

All of these systems focus on the widget level of user interfaces i.e. edit boxes, text captions, buttons and sliders. Any new iconic (widget) forms must typically be programmed using the underlying implementation language, or can not be supported at all. Thus these user interface builders are inappropriate for more general editing interfaces with complex icons, such as the graph construction used in CernoII and SPE, and geometric drawing applications. Systems like Unidraw [16] and EDGE which support the definition of such editors are generally limited to textual, specialisable frameworks, thus making it difficult to visualise the resultant user interface while refining its definition.

Haarslev and Möller's TEX-like layout language for visualizing CLOS objects provides an abstract method of specifying icon and glue structure [9]. It also provides a mechanism for mapping application data to visualization icon data. It is completely text-based, however, making viewing of the visualisation difficult during construction. Iconographer [5] takes data from files and converts it into application objects by a user-specified filter. Icon attributes are linked to application data attributes using a graphical switchboard metaphor. There is no tool for easily defining new icons or specifying their attributes, which limits its usefulness.

DSL [6] provides a text-based specification language for defining icon structure based on application object attributes. Attribute values are obtained from application objects via hierarchical abstractors, allowing very complex visualisations of complex application structures to be built. DSL includes constructs for composing attribute values into lists, vertical and horizontal alignment of lists, padding glue, edit boxes, buttons, anchor points for glue connections, and dialogue, menu and window components. This textual notation is abstract, flexible and

adaptive to application object changes. Like Haarslev and Möller's language, however, it is difficult to build complex icons interactively due to a lack of visual feedback during construction. We have designed the Skin notation based on the components of DSL, and developed a visual programming environment for Skin to address this fundamental problem of interface construction.

3. Skin overview

Skin is a visual functional language, using an icon and connector model for program construction. The language includes the following features:

- Skin functions define the layout of a user interaction component.
- A simple interface to the underlying system is assumed.
- Absolute and relative coordinates are avoided by the use of fill and alignment primitives, and subcomponent sizes (eg text box lengths) adapt to the actual size of run-time data values they visualize.
- Skin functions can have multiple clauses selected by pattern matching on parameters. This, together with list processing capabilities, allows flexible component construction based on actual parameter values.
- Higher order functions may be defined and used.
- User interaction is handled via primitives which sensitise regions to interactive events.
- All skin functions, including primitives, have associated (editable) default values for their parameters. These are used to generate prototype examples of interface components to provide both immediate feedback to the programmer and to automatically construct meaningful icons for user defined Skin functions.
- The visual language maps in a straightforward manner to DSL, allowing programming in either visual or textual modes.

4. Skin environment

Fig. 2 shows a screen dump of the Skin programming environment in use. Two windows are shown. The larger is a visual programming window, in which a function is being constructed (centre) using a palette of functions (top). Functions have input parameters at the top (solid line with pin underneath) and result at the bottom (solid line with two pins above and one below). The smaller window is a textual view of the function being constructed in the visual programming window.

Skin functions are visually constructed by direct manipulation. Palette icons are selected by mouse click. The *Viewer* primitive is currently selected as indicated by highlighting. Dragging from the result (lower) pin of an existing (non-palette) icon to an empty location clones the current palette icon and wires the output pin of the

existing icon to an input pin of the new icon. Clicking elsewhere in the drawing window clones the current palette icon. Pins of existing icons may also be wired together by dragging from an output pin to an input pin.

Textual views may be edited using free-form editing. Multiple visual and textual views can be created to edit the same or different functions. Visual views are kept consistent with one another. Textual to visual view consistency is implemented using the MViews update-record-based consistency management mechanism [8]. This allows programmers to edit either type of view and have changes propagate to all other affected views. Unlike FormsVBT, changes to the graphical view which may not be directly translated to changes in the textual view and vice versa can be supported, overcoming a disadvantage of the formsVBT approach [9].

A brief description of a selection of the Skin primitives can be found in the Appendix. In the next section examples of Skin functions and expressions are used to illustrate the major features of the language.

5. Skin by example

5.1 A simple function

Fig. 3 shows a Skin function. This takes two arguments and constructs a horizontal list containing a textually formatted version of the first argument, a block of white space, an alignment marker (see below) and a horizontal bar, the length of which is specified by the value of the second argument.

The text formatter, at the left, takes any value and constructs a printable version of that value which is then formatted, in this case as plain text of the default window font (variants for other styles and fonts are available). The bar constructor, at the right, takes an integer argument and constructs a bar of size proportional to the argument.

The white space constant uses a variant of the palette icon cloning mechanism to construct a version of the white space primitive with input argument (size) set, via dialogue, to a constant value.

The alignment primitive is used to line up icon components across multi dimensional lists. An example of its application is shown in Fig 4 (see later).

A horizontal list constructor is used to gather the above elements together horizontally in sequence. List constructors have a variable number of input pins. As wires are attached to existing pins of such an icon, additional pins are created. The elements in the list are ordered, with earlier elements to the left and later elements to the right.

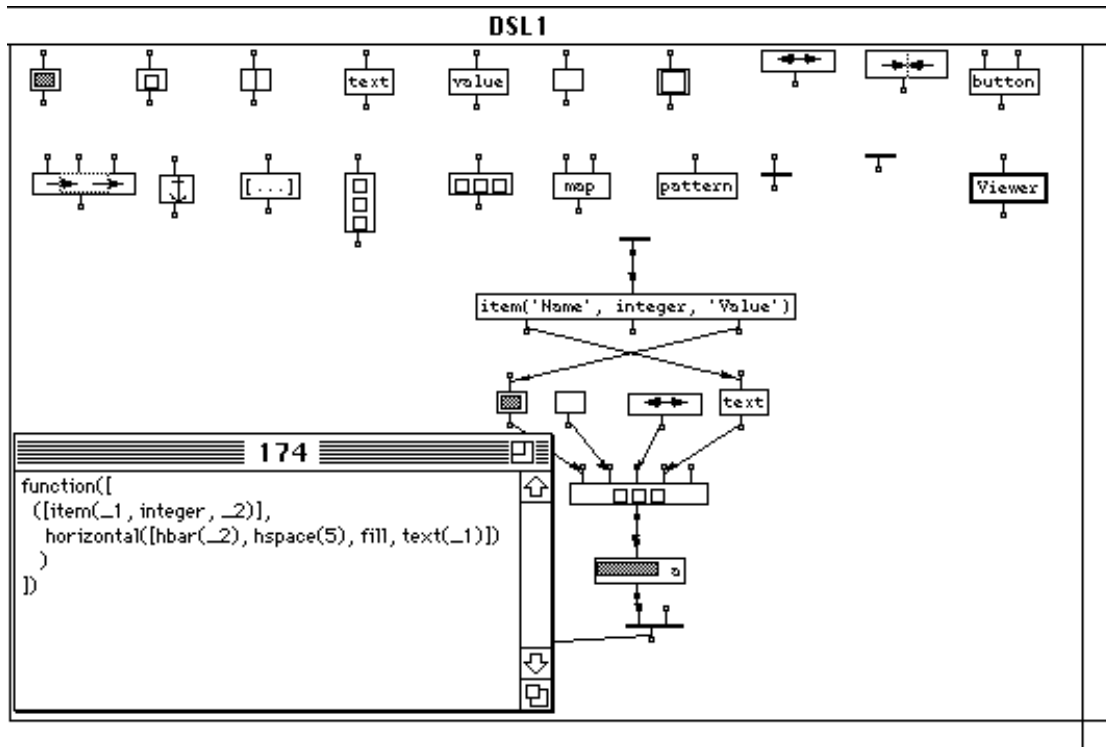


Fig. 2 Skin programming environment

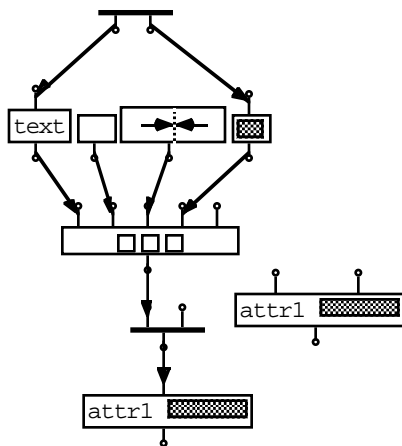


Fig. 3 Example Skin function

Attached to the function result pin is a viewer icon. Viewers are used to render a prototype version of the Skin fragment to which they are attached. In this case, the result of a prototype application of the function is displayed. To allow such prototypes to be constructed, each function input (including primitives) has associated (editable) default values. If nothing is attached to a function input, the default value is used whenever the function result is needed. In the example of Fig. 3, the function parameters have associated default values of *attr1* and 30 (defined by dialogue) resulting in the prototype icon shown.

Viewers show the final appearance of a prototype application of a function. A simple extension would be to also provide views showing the "hidden elements", such as white space and alignment markers, in a similar manner to that taken in FormsVBT [2].

To the bottom right of the function is a new palette icon constructed from the function definition. This may be cloned in the same way as primitives to create applications of the function. An important aspect of the language is that sensible icons for user defined functions are constructed automatically using the same approach as is used by viewers: ie constructing a prototype application using the argument defaults. This is in contrast to other visual programming languages which either have similar looking icons, distinguished by textual annotation (eg Prograph [4]) or require programmers to hand-craft icons (eg Fabrik [10]). For functions with large prototypes, scaling may be needed to produce a usable icon.

The Skin approach allows interface programmers to ignore details such as absolute co-ordinates for the resulting icon and its subcomponents as these are instantiated by Skin when the icon is actually drawn. This contrasts with most interface builders, such as Fabrik and FormsVBT, where the position of components must be specified by absolute co-ordinate values or direct manipulation. Text field length based on actual attribute

values, white space adjusted accordingly and alignment operators applied for actual text field lengths and white space size allows very adaptive icons to be specified. When the application attribute values change, Skin adaptively changes the icon layout and redisplay it.

5.2 Function application

Fig. 4 shows 2 applications of the function defined in Fig. 3. The actual arguments to the function applications are, in this case, constant primitives. The results are combined using a vertical list constructor, creating a constant Skin expression displayed using a viewer. The alignment primitive included in the function acts to line up the left hand edges of the bar components of the horizontal lists by padding out the previous subcomponent, if necessary, with white space.

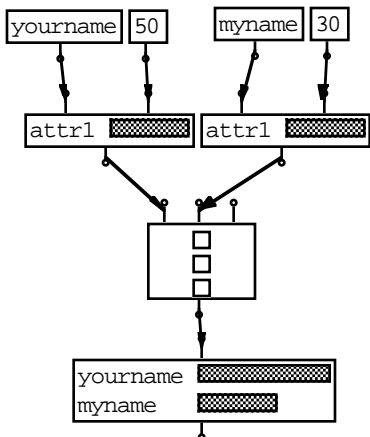


Fig. 4 Application of the function of Fig. 3

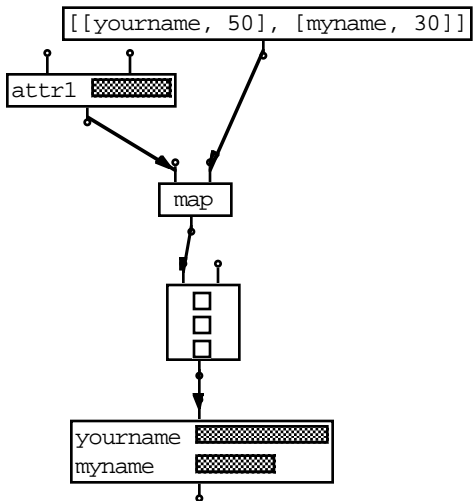


Fig. 5 Use of map higher order function

5.3 Higher order functions

Fig. 5 shows the same result as Fig. 4 achieved using the *map* higher order function. *Map* takes a function as its

first argument and applies it to each element of the second, list, argument. The result is then formed into a vertical list using a vertical list constructor. Higher order functions may also be defined by the user.

5.4 Multi-clause functions and pattern matching

Multi-clause functions permit very adaptive component definition as they allow different Skin expressions to be constructed depending on the form of the parameters. When a multi-clause function is applied, one of its clauses is selected for execution using pattern matching on actual parameters. Fig. 6, for example, shows a function with two clauses. Each clause takes a single input, which is matched against a pattern. The pattern for the left hand clause is the tuple *item(N,int,Val)*. For this clause to be used, the actual argument must be a 3-tuple with functor *item*, and second element *int*. The first and third elements can match anything¹. The pattern for the second clause is also an *item* triple, but with second element the value *ref*.

The order in which clauses are attempted is determined by their connection to the result icon. This icon, like the list constructors, has an expandable number of input pins. Each "input" corresponds to the "result" of a clause for that function. Extra clauses are added by wiring to the spare right hand pin. The Skin environment allows clauses to be constructed either in the same window (as is the case with the function of Fig. 6) or in separate windows. In either case, the result icon clearly indicates the position in the clause "list" of the function. When the function is applied, clauses are attempted from left pin to right pin until a pattern match succeeds, in which case the function body is evaluated.

Patterns serve both as clause selectors and also as argument "decomposers". They have multiple outputs, one for each element of the pattern. These may be used, as in the function of Fig. 6, to access the values of the pattern elements.

Icon and viewer construction for multi-clause functions is quite straightforward. The default inputs for each case are gathered up into a list (in clause order) which is mapped by the multi-clause function with the result being formed into a vertical list. This is illustrated by the viewer and palette icons of Fig. 6, where the separate prototype cases for each clause are clearly visible. In some cases, a vertical list of all clause prototypes may not, for reasons of size or elision of detail, be the best approach to icon (or viewer) generation.

¹A Prolog-like syntax is used for patterns. Although described as pattern matching, the patterns may in fact be quite complex predicates acting as guards for the clause. The form of a pattern is specified via dialogue on cloning from the pattern palette icon.

Another possibility is selection of the prototype for only one case, perhaps with other case prototypes available through a popup menu.

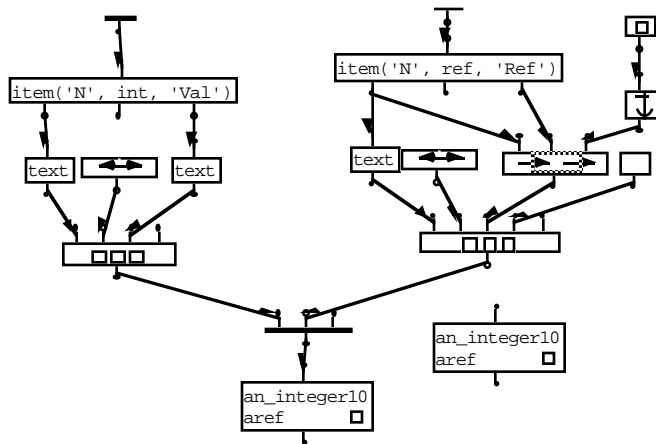


Fig. 6 Multi-clause function with pattern matching

5.5 Specification of connectors

The second clause of the function in Fig. 6 illustrates specification of inter-component connectors using a combination of *connector regions* and *anchors*. Connector regions have three arguments: a name for the region, a reference (possibly nil) to a program object, and a Skin expression. The connector region primitive has an arrow entering and leaving a dashed border box.

Connector regions serve two purposes. Firstly, if a component associated with the object referenced by the second argument is visible, a connector line is drawn between the two components². A variety of line styles (plain or greyed, arrowed ends or not) are available. Secondly, connector regions specify the location of the start and end points of connectors. These are specified via anchor regions contained within the picture associated with the third argument to the connector region.

Multiple anchors may be specified for a connector region. When a connector is made, the two closest anchor regions are chosen and the line is drawn between the midpoints of each. Each anchor takes a Skin picture as its argument. In Fig. 6, the connector region consists of a single anchor region (specified by the anchor primitive which has obvious shape) which covers a small square box. If either display associated with a connection is moved or changes shape, the connection is redrawn, possibly using different anchor regions. Fig 1(a) shows examples of inter-component connectors used to visualize

² Note that the reference may be to an object, in which case the connection is to the other display as a whole, or an object-name pair, in which case the connection is to a subregion of the other display associated with that name.

object references and method calls in Cerno. These components include anchors at both sides of each attribute subcomponent.

5.6 System interface

Skin functions assume a very simple object oriented interface with the underlying systems they are visualising. User interface components are constructed by first constructing a display object. This object executes a Skin function to construct the visible interface component. The display object must be able to provide a list of *item(Name,Type,Value)* triples to the executing Skin function, which the function can make use of to construct the component. Any changes to elements in the list trigger reconstruction of part or all of the interface component. User interaction events, such as button presses, are signalled to the display object via method calls.

The Skin system interface primitive accesses and selects required values from the item list. Fig. 7 shows an example of the use of this primitive in the construction of a nullary function. This function displays all elements of the item list which are of integer or ref type using the function of Fig. 6, arranging the result as a vertical list.

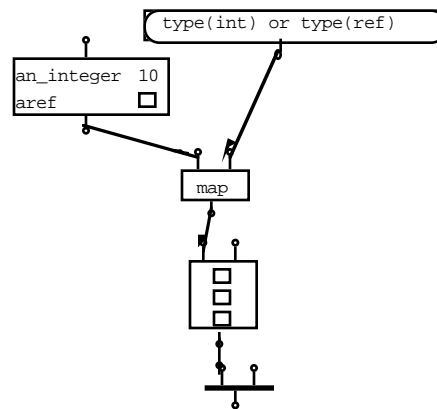


Fig. 7 System interface example

5.7 User interaction

User interaction with Skin functions is achieved by applying a side effect function that specifies a region as being sensitive to user input.

The button primitive takes two arguments: a name to give the button, and a picture for the button. The left hand function of Fig. 8 is a nullary function to create an Ok button, with name 'ok' and button picture a double box containing the text "Ok" padded with some white space. When incorporated into an interface component, clicking in the picture associated with the button results in a call to the *button* method of the underlying display object with parameter *ok*, being the name of the button. It is up to the underlying application to interpret the button press in an

appropriate way (such as removing the interface component in the case of the Ok button).

Edit boxes take a name and an initial value, and construct a textually editable region filled with the initial value. When edited, a call is made to the *edit* method of the underlying program object with two parameters: the name of the edit region, and the newly entered value. Again, it is up to the application to interpret the method call in an appropriate way. The function on the right hand side of Fig. 8 incorporates an edit box and an Ok button to construct a dialogue box for editing a feature value. The parameters are a name for the feature and the initial value of the feature. White space is used to pad the dialogue box out vertically, and the fill primitive is used to position the Ok button at the right hand side of the dialogue box. The viewer at the bottom shows a prototype of the resulting dialogue box.

Other types of input component can be handled in a similar way. Popup menus, for example, could be constructed using a button; ; selection causes a new list component to be created and overlaid on top of the button.

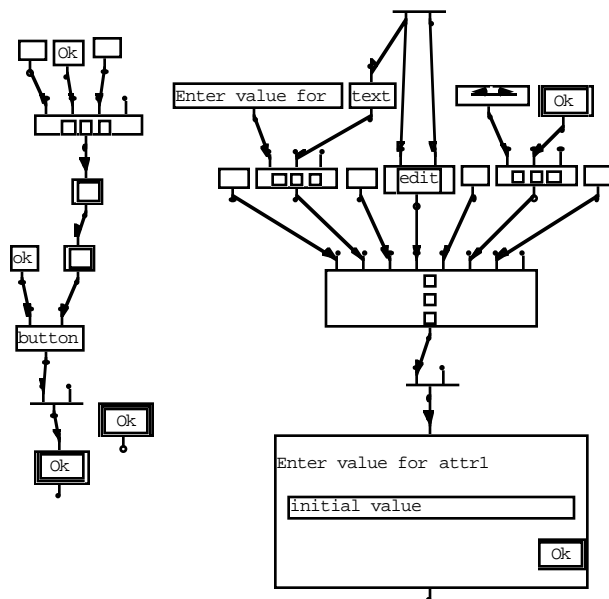


Fig. 8 Button and edit boxes

6. Implementation

DSL, the textual analogue to Skin, was developed as part of the Cerno program visualization system [6] where it is used to construct user interface components depicting the state of objects or abstractions of collections of objects such as the ones in Figs 1(a) and (b). Cerno uses a multi-layered architecture to perform program visualization: application objects are traced using *abstractor* objects which gather information from the traced objects (attributes and values plus active method calls). The

resulting list of values may be used by display objects (incorporating DSL functions) to construct interface components (such as those of Fig 1(a)) or passed through one or more other levels of abstractor (arranged in a hierarchy) abstracting away detail in the process (to construct, for example, the list views of Fig 1(b)).

The visual Skin programming environment was constructed by specialising Cerno (itself a specialisation of the MViews programming environment framework [8]) bootstrapping from DSL. Skin programs are represented in the environment as a graph of objects, one object per visible icon. Each object includes an attribute defining its DSL function, a method for cloning the icon (for palette icons), and a method for constructing an instantiation of the function from its actual arguments. Abstractors for various types of icon (standard, expandable pin, viewer) were constructed, together with a standard display component capable of rendering any type of icon using Skin itself. The only additions to DSL required were icons for input and output pins.

7. Conclusions and future work

We have described Skin, a visual functional language for defining flexible user interface components. The language combines expressive power with the ability to view concrete instantiations of the components being defined. The instantiations can, in turn, be used to construct meaningful icons for user defined functions.

Current work is aimed at construction of additional primitives, particularly for interactive components; improved performance (the current implementation is doubly interpreted); the addition of a FormsVBT-like direct manipulation editor for individual cases. Future work includes reuse of Skin graphs for visualising execution of Skin functions, and addition of typing and type inference.

Skin addresses the problem of specifying layout of interface components, including interactive components such as buttons and edit boxes. However, the semantics of interaction, ie what happens following the edit or button method calls, is not specified in Skin. We are working on a separate visual language for specifying this using and extending the Lean Cuisine [1] and LC+ [15] approaches. A visual specification of Cerno abstractors is also being developed. This will allow a connection between the Skin layer and application object layer to be defined, similar to the approach of Iconographer [5], but with hierarchical abstractor support.

Acknowledgements








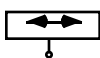
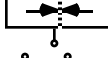


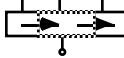





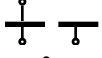

John Hosking acknowledges the financial support of the University of Auckland Research Committee in

performing this work. This work was done while John Hosking was a visitor to the Software Verification Research Centre at The University of Queensland and he acknowledges the assistance of that Centre in preparing this paper.

References

- [1] Apperley, M.D. & Spence, R. 1989: Lean Cuisine: A Low-Fat Notation for Menus, *Interact. with Comput.*, 1, 1, 43-68.
- [2] Avrahami, G., Brooks, K.P., Brown, M.H. 1989, A Two-View Approach to Constructing User Interfaces, In *ACM Computer Graphics*, Vol. 23, No. 3, July 1989, 137-146.
- [3] Brown, M.H. 1991: Zeus: A System for Algorithm Animation and Multi-View Editing, In *1991 IEEE Symposium on Visual Languages*, 4-9.
- [4] Cox, P.T., Giles, F.R., Pietrzykowski, T. 1989: Prograph: a step towards liberating programming from textual conditioning, *1989 IEEE Workshop on Visual Languages*, 150-156.
- [5] Draper, S.W., Waite, K.W., Gray, P.D., "Alternative Bases for Comprehensibility and Competition for Expression in an Icon Generation Tool", in *Human-Computer Interaction - INTERACT '90*, D. Diaper, D Gilmore, G Cockton, & B Shackel, eds., Elsevier Science (North Holland), 473-477.
- [6] Fenwick, S. 1994: A Visual Debugger for Object-Oriented Programs, MSc Thesis, Department of Computer Science, University of Auckland,.
- [7] Grundy, J.C., and Hoking, J.G. 1993: Integrated object-oriented software development in SPE, Proceedings of the 13th NZCS Conference, Aucland, New Zealand, August 1993, 465-478.
- [8] Grundy, J.C., and Hosking, J.G. 1993: The MViews framework for building visual programming environments, Proceedings of the 1993 IEEE Symposium on Visual Languages, 1993, 220-224.
- [9] Haarslev, V., Möller, R. 1992: Visualization and Graphical Layout in Object-oriented Systems, *Journal of Visual Languages and Computing* 3(1), 1-23.
- [10] Ingalls, D., Wallace, S., Chow, Y.Y., Ludolph, F., Doyle, K., 1988: Fabrik: A Visual Programming Environment, Proceedings of OOPSLA '88, 176-189.
- [11] Linton M.A., Vlissides J.M., Calder, P.R. 1989: Composing user interfaces with Interviews, *COMPUTER* 22 (2), February 1989, 8-22.
- [12] Mashita, K., Matsuoka, S., Takahashi, S. 1992: Declarative Programming of Graphical Interfaces by Visual Examples, Proceedings of USIT '92, 107-116.
- [13] Myers, B.A. 1987: Creating User Interfaces by Demonstration, PhD Thesis, University of Toronto.
- [14] Paulisch, F.N., Tichy, W.F., EDGE: An Extensible Graph Editor, In *Software - Practice and Experience*, Vol. 20, No. S1, June 1990, pp. S1/63-S1/88.
- [15] Phillips, C.H.E. 1993, The Development of an Executable Graphical Notation for Describing Direct Manipulation Interfaces, PhD Thesis, Massey University.
- [16] Vlissides, J.M. Generalized Graphical Object Editing, PhD Thesis, Stanford University, CSL-TR-90-427, 1990.

Appendix: Selected Skin Primitives

	horizontal or vertical bar, size specified by argument
	square box, argument is background pattern
	horizontal or vertical line, argument is thickness
	text formatter, argument is any value, style set by dialogue
	value, just passes argument through to result. Usually used to construct constants.
	horizontal or vertical white space, argument is size of space
	framebox, argument is Skin picture to frame
	fill, pad out with horizontal or vertical white space in multi dimensional list
	align corresponding elements in multi dimensional list
	button, arguments are button name and picture
	edit box, arguments are name of edit region and initial value
	connector region, arguments are name, destination object, and picture
	anchor region, potential start or end point for enclosing connector region argument is picture
	horizontal list constructor, expandable no of args, form args into horizontal list (lines -> vertical lines, white space -> horizontal white space, bars -> horizontal bars, etc)
	map, apply first argument (a function) to each element of second argument (a list)
	pattern matcher, must be attached to parameter pin, pattern specified by dialogue
	function result and function parameter pins
	viewer, render prototype view of argument, pass argument directly to output
	system interface, selection criteria specified by dialogue