To appear in the Proceedings of OZCHI'95, Wollongong, Australia, Nov 28-30, 1995.

# COVER YOUR SELF WITH SKIN

*John G. Hosking*[†], *Warwick B. Mugridge*[†], *Stephen Fenwick*[††] *and John C. Grundy*[†††]

[†]Department of Computer Science
University of Auckland
Private Bag 92019, Auckland,
New Zealand
{john, rick}@cs.auckland.ac.nz

[††]Department of Computer Science
Australian National University
Canberra, ACT
stevef@cs.anu.edu.au

[†††]Department of Computer Science
University of Waikato
Private Bag 3105, Hamilton,
New Zealand
jgrundy@cs.waikato.ac.nz

## ABSTRACT

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.

## INTRODUCTION

The definition of user interface components can often be a tedious process, involving trial and error construction of the layout of components using a textual definition language [10]. More rapid layout definition 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.

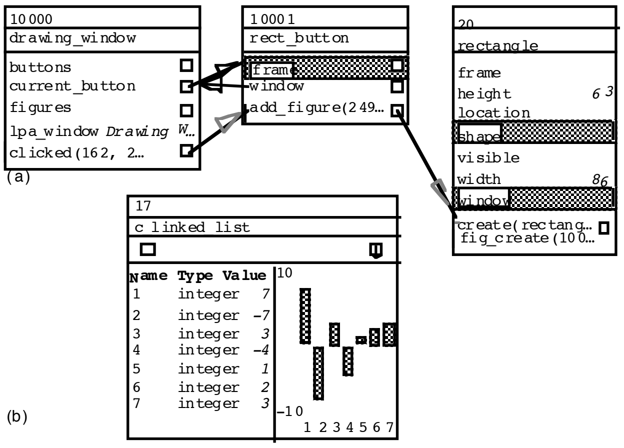


Fig. 1 Examples of flexible user interface components

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 indicates 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. Eg clicking in the square box of an object reference attribute causes that reference to be expanded as a new interface component.

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), and this specification should cope with changes to the component (eg addition of extra attributes in the list) as the program executes. 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, ie ones that adapt their composition and layout automatically as the needs of the underlying application change, as well as more conventional dialog box, menu, and window components.

Many user interface toolkits, such as Fabrik [8] and Interviews [9], provide 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. Such complex icons, and editors to manipulate them, can be defined using programming techniques. However, these use textual, procedural languages, and the resulting icon definitions are often inflexible, and difficult to maintain and understand due to their low-level, complex details [7].

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

## RELATED WORK

Traditional UI construction can be done via toolkits, frameworks and UIMSs [9]. Interviews [9] 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 interface components. Their programs must be run to test the interface appearance and functionality. Interface builders, like Peridot [11] or FormsVBT [2], allow specification of interface components by direct manipulation. Interface semantics are programmed textually. FormsVBT keeps the graphical appearance of the interface consistent with a textual description, allowing specification in whichever form is most convenient. A problem with this approach is the cluttering of the direct manipulation view for complex interfaces [7]. FormsVBT also provides a view showing the resultant user interface [3]. This can be interacted with to test the interface.

Prograph [4] includes a graphical interface builder and allows the interface semantics to be programmed visually. Fabrik [8] uses a similar dataflow metaphor but its data entities directly correspond to user interface components. Fabrik diagrams are live (even when incomplete), similar to FormsVBT. The interface can thus be incrementally tested during construction. A disadvantage is the use of absolute co-ordinates for positioning interface components, making the system less adaptive to application change.

EDGE [12] supports generation of graphical editors for graph-based structures from a textual specification of the graph. Graph node and edge icons must, however, be programmed to achieve complex icons. Trip 3 [10] allows 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. New iconic (widget) forms must typically be programmed using the underlying implementation language, or can not be supported at all. Thus these systems are inappropriate for more general editing interfaces, such as the graphs used in CernoII, and geometric drawing applications. Systems like Unidraw [14] and EDGE which support the definition of such editors are usually based on textual, specialisable frameworks, making it hard to visualise the resulting user interface while refining its definition.

Haarslev and Möller's $\mathrm{T_EX}$-like layout language provides an abstract way of specifying icon and glue structure [7], plus a mechanism for mapping application data to icon data. It is, however, text-based, making 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.

## SKIN OVERVIEW

Skin is a visual functional language, using an icon and connector model for program construction. 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.

All skin functions have (editable) default values for their parameters. These are used to generate prototype examples of interface components to provide 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. User interaction is handled via primitives which sensitise regions to interactive events.

## 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 specified (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 same function.

Skin functions are visually constructed by direct manipulation. Dragging from the result (lower) pin of an existing (non-palette) icon to an empty location clones the currently selected 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.

Textual views may be edited using free-form editing. Multiple visual views can be created to edit the same or different functions with views being kept consistent with one another. This allows programmers to edit any type of view and have changes propagate to all other affected views. Changes may also be made to the graphical view which cannot be directly translated to changes in the textual view and vice versa, overcoming a disadvantage of the formsVBT approach [7]. In the next section examples are used to illustrate the major features of the language.

## SKIN BY EXAMPLE

### 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, with length specified by the second argument's value. The text formatter (text) takes any value and constructs a printable formatted version. The bar constructor (▨) 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 (see Section 5.2). 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.

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

The palette of functions may be extended by the user. To the bottom right of Fig. 3 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 feature is that sensible icons for user defined functions are constructed automatically using the same approach as is used by viewers. In constrast, other visual programming languages either have similar looking icons, distinguished by textual annotation (eg Prograph [4]) or require programmers to hand-craft icons (eg Fabrik [8]).

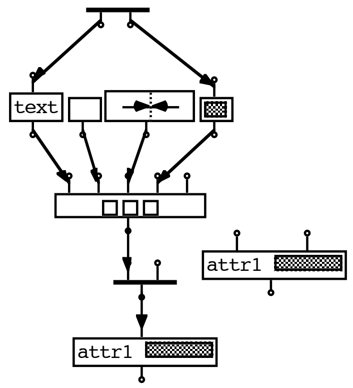
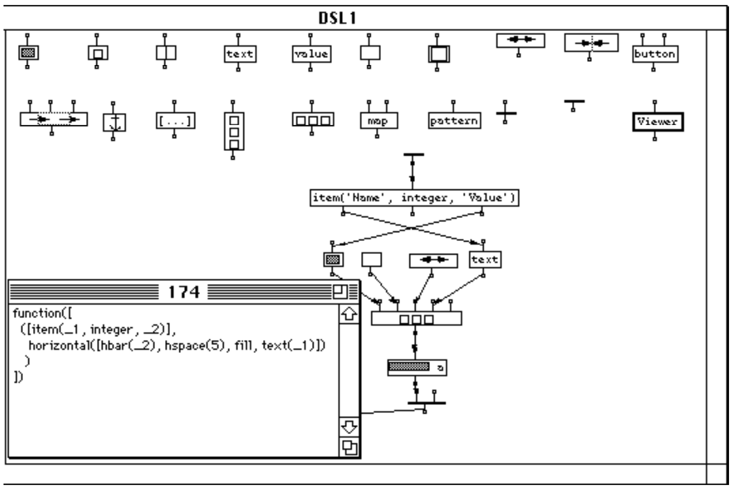


Fig. 3 Example Skin function

The Skin approach allows interface programmers to ignore details such as absolute co-ordinates for the resulting icon and its subcomponents as they are instantiated by Skin as the icon is drawn. In contrast, most interface builders, such as Fabrik and FormsVBT, need component positions to be specified by absolute or relative co-ordinate values or by 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 application attribute values change, Skin adaptively changes the icon layout and redisplays it.



```
function([
 ([item(_1, integer, _2)],
  horizontal([hbar(_2), hspace(5), fill, text(_1)])
 )
])
```

Fig. 2 Skin programming environment

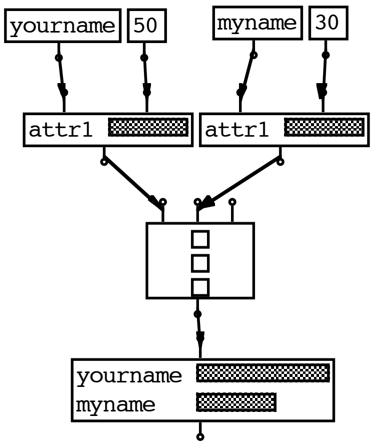


Fig. 4 Application of the function of Fig. 3

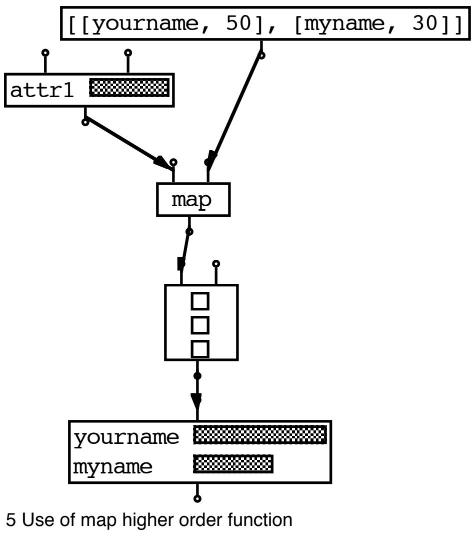


Fig. 5 Use of map higher order function

**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 and 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.

**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.

**Multi-clause functions and pattern matching**
Multi-clause functions allow different Skin expressions to be constructed depending on the form of the parameters. Pattern matching on actual arguments is used to select between clauses. Fig. 6, for example, shows a function with two clauses, one matching tuples representing integer attributes and the other matching reference attributes. 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 anything[1]. The pattern for the second clause is also an *item* triple, but with second element the value *ref*. Patterns can also be used to decompose complex arguments. They have multiple outputs, one for each element of the pattern.

The result icon, like the list constructors, has an expandable number of input pins. Each "input" corresponds to the "result" of a clause for that function. When the function is applied, clauses are attempted from left pin to right pin until

[1]A Prolog-like syntax is used for patterns. 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.

a pattern match succeeds, in which case the function body is evaluated.

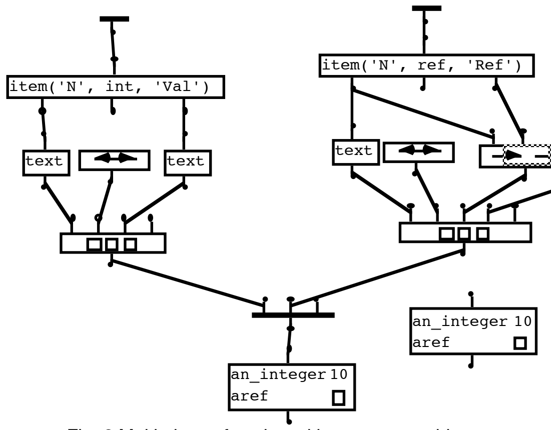


Fig. 6 Multi-clause function with pattern matching

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.

## Specification of connectors

The second clause of the function in Fig. 6 shows the specification of inter-component connectors using *connector regions* ( )and *anchors* ( ). Connector regions have three arguments: a region name, a reference (possibly nil) to a program object, and a Skin expression. 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 components. 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 a line is drawn between the midpoints of each.

In Fig. 6, the connector region consists of a single anchor region 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 object references and method calls in Cerno. These components include anchors at both sides of each attribute subcomponent.

## System interface

Skin functions assume a very simple object oriented interface with the underlying systems they are allowing the user to interact with. 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 by method calls. The interface between diaply objects and the underlying system is application dependent. The approach used in one application is described in the following section.

The Skin system interface primitive ( )accesses and selects required values from the diplay object's 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.

## User interaction

User interaction with Skin functions is achieved by applying a side effect function that specifies a region to be 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 box containing the text " Ok ". When incorporated into an interface component, clicking in the button's picture results in a call to the *button* method of the underlying display object with parameter *ok*. It is up to the underlying application to interpret the button press in an appropriate way.

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. The fill primitive is used to position the Ok button at the right of the dialogue box. The viewer (bottom) shows a prototype of the resulting dialogue box.

The style of interaction with Skin user interface components (ie buttons requiring a mouse down event) is thus hard coded in the side effecting primitives. This is a current weakness of our approach, as it limits the level of interaction with which the interface designer can work with in constructing interface components.

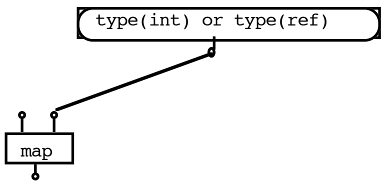


Fig. 7 System interface example

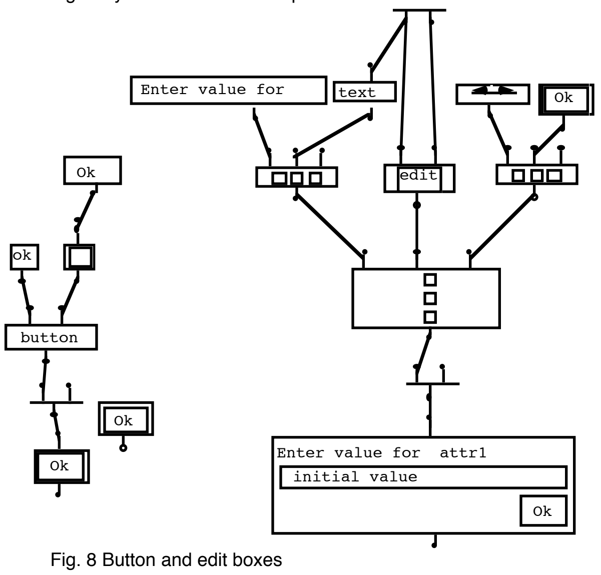


Fig. 8 Button and edit boxes

## 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. *Abstractor* objects gather information from traced application objects and pass that information to display objects. Cerno's display objects take the resulting list of values and execute an embedded DSL function to construct interface components (such as those of Fig 1(a)). Aabstractors may also pass information to one or more other levels of abstractor before passing information on to a display object to construct more abstract views, such as the list views of Fig 1(b)).

The Skin programming environment was constructed by specialising Cerno. 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.

## 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 and concrete instantiations of the components being defined. The instantiations can, in turn, be used to construct meaningful icons for user defined functions.

Skin addresses the problem of specifying layout of interface components, including interactive components such as buttons and edit boxes. Extensions planned include associational primitives, that permit subcomponents to be associated in a relational way (left, right, etc) to resizable parent components (currently parent components sizes are completely determined by the composition of the child component sizes).

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+ [13] approaches (particularly the semantic definition extensions of the latter). 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. We are also exploring extensions to Skin that allow new user interaction components (eg buttons) to be defined from more basic interaction operations (such as dragging and clicking).

## ACKNOWLEDEGEMENTS

John Hosking acknowledges the 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*, **23** (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., Hosking, J., and Mugridge, W.:, 1994: A visualization system for object-oriented programs, *Proc. TOOLS 15*, Prentice Hall, Sydney, pp 93-103.

[7] Haarslev, V., Möller, R. 1992: Visualization and Graphical Layout in Object-oriented Systems, Journal of Visual Langages and Computing, **3** (1), 1-23.

[8] Ingalls, D., Wallace, S., Chow, Y.Y., Ludolph, F., Doyle, K., 1988: Fabrik: A Visual Programming Environment, Proceedings of OOPSLA '88, 176-189.

[9] Linton M.A., Vlissides J.M., Calder, P.R. 1989: Composing user interfaces with Interviews, COMPUTER, **22** (2), February 1989, 8-22.

[10] Mashita, K., Matsuoka, S., Takahashi, S. 1992: Declarative Programming of Graphical Interfaces by Visual Examples, Proceedings of USIT '92, 107-116.

[11] Myers, B.A. 1987: Creating User Interfaces by Demonstration, PhD Thesis, University of Toronto.

[12] Paulisch, F.N., Tichy, W.F., EDGE: An Extensible Graph Editor, In *Software - Practice and Experience*, **22** (S1), June 1990, pp. S1/63-S1/88.

[13] Phillips, C.H.E. 1993, The Development of an Executable Graphical Notation for Describing Direct Manipulation Interfaces, PhD Thesis, Massey University.

[14] Vlissides, J.M. Generalized Graphical Object Editing, PhD Thesis, Stanford University, CSL-TR-90-427, 1990.