A Declarative Formalism for Specifying Graphical Layout

Volker Haarslev*
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304, USA
haarslev@parc.xerox.com

Ralf Möller
University of Hamburg, AI Laboratory
Bodenstedtstr. 16,
moeller@rz.informatik.uni-hamburg.dbp.de

Abstract

This paper describes a new approach to specifying graphical layouts of arbitrary objects, which is based on a TEX-like notation. Our simplest scheme offers specifications similar to TEX's box-and-glue metaphor. Size and position of boxes and glue can be specified by constraints. Advantages of this TEX-like formalism are its expressiveness, user-predictable layouts, and efficient implementation schemes for the underlying layout algorithms. We extend and generalize this forms-oriented scheme for specifying advanced graphical user interfaces (e.g. CLOS class browser).

1 Introduction

This paper discusses a new approach combining structural and conceptual visualization techniques, which is based upon TEX-like layout specifications. We implemented a prototyping environment consisting of a set of extensible fundamental components which offer varying degrees of support. It is implemented in Macintosh Allegro Common Lisp and based upon the PCL implementation of the Common Lisp Object System [2, 11] (CLOS). Our visualization techniques are based on the CLOS meta-object protocol and require no modifications to the application systems which are selected for visualization. We support multiple views as well as controllers for manipulating the application's data structures. A detailed discussion of these meta-level techniques is presented in [8, 9].

In order to verify the feasibility of our approach we applied our visualization techniques to several application domains such as CLOS debugging, graphical editing, constraint systems, line routing, and concurrent logic programming (see [15, 9] for details).

Conceptual visualizations of programs are mostly created by hand. This hand-design is basically caused by the fundamental problem that geometrical and graphical information necessary to create suitable visualizations cannot automatically be derived from corresponding data. The major problem is to define interesting events which should be visualized. But very often interesting events are only indirectly reflected by algorithms. We refer to [4] for a detailed discussion of these problems.

Structural visualization uses program and data structures to generate relevant geometrical information. An important problem related to structural interpretation is that conceptual information about data can only indirectly be derived (eg. from naming of identifiers). A very common approach to structural visualization is to guide the visualization process by underlying programming styles or computational models. Many approaches to visualizing imperative systems use flow charts or diagrams. The Transparent Prolog Machine [6] is an example for relational or logic systems. A more radical approach is presented by Pictorial Janus [10]. It defines complete visualizations of concurrent logic programs and captures static as well as dynamic information about these programs. There exist many approaches to visualizing data flow of functional systems, e.g. VIPEX [7] and Prograph [5].

Another important consideration — and the primary focus of this paper — is to allow flexible schemes for aesthetically laying out combinations of units. With respect to forms-oriented user interfaces allocation of space and position is mostly constrained by the space globally available. Graphical interfaces usually also add local constraints. A typical application is a browser generating net-like representations of rule sets, classes, or objects. The spatial allocation of nodes may depend on adjoining nodes or the topology of edges (e.g. in order to avoid line crossing or long winding paths). This problem is addressed by many constraint-oriented systems. ThingLab I [3] and II [14] are examples for describing layout of graphical objects with constraints. [16] also presented a toolkit using constraints and active values. In contrast to constraint-oriented approaches we decided to provide a simpler but more compact and predictable notation for specifying layout. Furthermore, our approach has the advantage that it requires only $2n + \log(n)$ steps, be n the number of boxes. Thus, our algorithm has a computational complexity of $O(n)$.

The remainder of this paper is structured as follows. The next section introduces our TEX-like specifications. Section 3 generalizes our notion of box items and introduces a mechanism for specifying references between box items. The next section explains our box layout algorithm in more detail. Section 5 compares our approach with related work. This paper concludes with a summary and a discussion of future work.

2 Layout Specifications

We adopted the "box-and-glue" metaphor of TEX [12] for specifying layout of objects. Layouts are composed of a set of rectangular regions, so-called *boxes*. Laying out boxes and positioning objects are associated with corresponding

* New address: University of Hamburg, Computer Science Department, Bodenstedtstr. 16, D-2000 Hamburg 50, FRG, haarslev@rz.informatik.uni-hamburg.dbp.de

```
(let ((left-table (make-dialog-item ...))
      (right-table (make-dialog-item ...))
      (graph-view (make-layout-view ...)))
 (make-layout-dialog :layout
  (:vbox (:width :filler :height :filler)
   (:hbox (:height 1/4 :width :filler)
    (:fbox () left-table) (:fbox () right-table))
   (:fbox () graph-view)))
 (setf (layout graph-view) ...))
```

Figure 1: Layout specification of Figure 2 (schematically).

box types. Our system offers a set of predefined layout algorithms and box types.

These algorithms are based on an abstract protocol for manipulating boxes and box items. Therefore, every object conforming to this protocol can be laid out and every (rectangular) region can be interpreted as a box. The protocol is implemented as a set of generic functions. Multi-object methods can be supplied for different kinds of boxes and items, which may represent position and size in different ways. The use of multi-object methods also has the advantage that layouts of objects may depend on their context. Our layout protocol also allows to specify whether the layout algorithm has to be reapplied if global constraints (e.g. by resizing of a window) have been changed.

2.1 Vertical and Horizontal Boxes

The fundamental scheme aligns boxes as a list of *horizontal* and *vertical* boxes. This layout technique has been found very useful for standard (forms-oriented) dialog windows (see Section 5 for a discussion of related work). A layout of a dialog window is specified as a combination of boxes with optional size specifications. Boxes may be arbitrarily nested. The size of boxes and the spatial relationship between them is expressed by an amount of glue or *filler* describing either a horizontal or vertical distance. Fillers can be specified as *fixed* (e.g. in pixel) or *variable*. Variable fillers depend on the space available to their enclosing box. We distinguish *relative* and *constrained* fillers. A relative filler is expressed as a fixed ratio to the size of its superior box. Constrained fillers can shrink (stretch) to a given lower (upper) limit. Default constraints are zero as lower limit and box size as upper limit. Several fillers as elements of the same box work together like springs. They share the available space and in general every filler claims the same amount of space which is only constrained by its lower and upper limit.

A vertical or horizontal box (`<box-type>` either `:vbox` or `:hbox`) is specified by the lisp form `(<box-type> (:width h :height v) box-item-1 ...)`. Its box items are laid out vertically resp. horizontally. If the size specification is omitted the width and height of a vertical or horizontal box are set to a filler with default constraints. In general this layout algorithm keeps the size of box elements unchanged. If elements require more space than available to their surrounding box they are allowed to extend beyond their box's boundaries. Boxes are also allowed to overlap one another. But this behavior is not always desired.

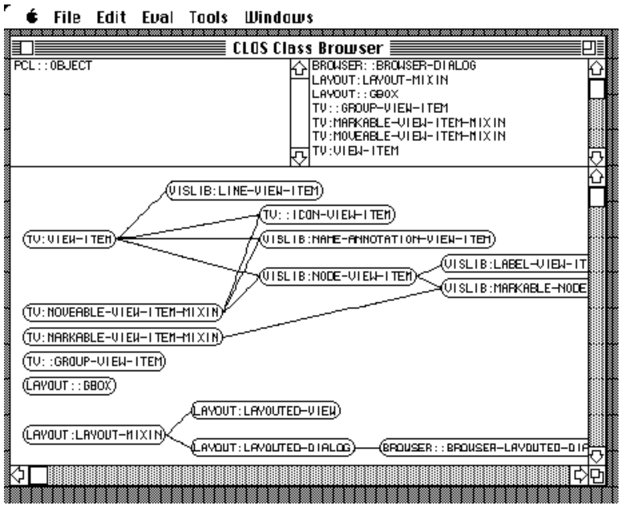

Figure 2: A DAG graph of a standard class hierarchy.

Therefore, we introduced a *frame box* (`:fbox`) which constrains the size of its element in order to match exactly the frame box size. A frame box contains only one box item: `(:fbox (:width h :height v) box-item)`.

2.2 Filler Specification

The complete form specifying a filler is `(:filler :min m :max n)`, `:min` and `:max` are optional. We defined `:filler` as short-form of `(:filler :min 0 :max <box-size>)`. It is also possible to define the (minimal/maximal) size of a box with respect to its elements. Then, the size of this box is set to the result achieved by laying out its elements and shrinking fillers to their lower limit (see [9] for more details). Thus, the box has a minimal size satisfying all lower bound constraints.

The Figures 1 and 2 show a layout specification and the resulting dialog window for a simple CLOS browser which displays a class hierarchy (DAG). The right table contains all direct subclasses of the class listed in the left table. The tables can be replaced by direct super resp. subclasses, scrolled, and shifted to focus on "interesting" classes.

The browser dialog is specified as a vertical box with `:filler` as width and height. Its first item is a horizontal box whose height is set to 1/4 of that of the vertical box. The second item is a frame box. It surrounds another box element (`graph-view`) which displays a graph of the selected class hierarchy. The horizontal box contains two scrollable tables which are enclosed by frame boxes. The height of these frame boxes is constrained by their surrounding horizontal box. Their width is not explicitly specified, therefore the default value `:filler` is chosen and half of the width of the horizontal box is assigned to each frame box (and its inferior table).

2.3 General Boxes

Apparently it is not reasonable to describe every layout with the box-and-glue metaphor. An obvious example is

```
(let ((upper-offset 10)
      (left-offset 10))
 (:vbox () upper-offset
  (:hbox () left-offset
   (:gbox
    (:dag
     roots
     #'successor-function
     depth
     #'map-function ...)))))
```

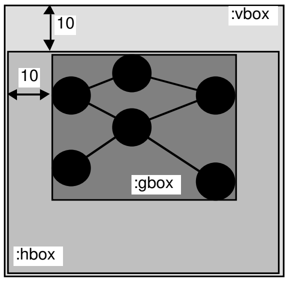

Figure 3: A general layout specification in combination with a box-style layout.[2]

a set of nodes arranged as a graph. Therefore, our layout language has the notion of a *general box*: `(:gbox (<layout-name> <arg-1>...<arg-n>))`. For instance, this box is used to specify the layout of directed acyclic graphs (DAGs) (see Figure 3).

In this example the items to be arranged are defined inductively by a set of roots, a successor function, and a maximal depth. The map function is used to compute the graphical representation of nodes (e.g. class objects for an inheritance graph). Position and size of the general box (`:gbox`) are defined implicitly by a closure rectangle around all graph items (see Figure 3). This rectangle defines a box which may be arranged using the box layout specifications already known.

One may also think of other arrangements of box items in a general box. We use the DAG example mentioned above in order to explain our protocol for supplying a new layout specification interpreter.[3]

```
(defmethod layout-spec-p-using-key
 ((key (eql ':dag))) t)
(defmethod parse-layout-spec-using-key
 ((key (eql ':dag)) layout-specs)
 "Returns (generated and) laid out :gbox items."
 (interpret-dag-layout layout-specs))
```

The layout name (e.g. `:dag`) of a general box form is used as a key to discriminate the corresponding layout interpreter method, the rest of the form is bound to the parameter `layout-specs`.

This extension scheme exploits that CLOS methods may not only be attached to objects (or their classes) but can be also discriminated on every Lisp object. Layout forms are represented as lists, i.e. layout descriptions can easily be manipulated (e.g. by pattern matching algorithms). [9] contains the complete EBNF syntax of layout specifications. The next section describes the implementation of the DAG graph in more detail.

3 Interaction Objects and Views

We introduce box items which are suited for more general applications and describe their application to our class browser which serves as a simple example for describing the basic features of our layout language. We refer to [15, 9] for a discussion of more sophisticated user interfaces.

3.1 Interaction Objects

Interaction objects represent box items of layout specifications. Interaction objects (e.g. all standard elements of the Macintosh Toolbox, graph nodes, graph edges) are instances of CLOS classes. We defined an additional interaction object, a so-called *view*. Views provide a framework for handling non-standard interaction components. These components are called *view items*. View items can be freely added to and removed from views. The interactive behavior of view items can easily be modified by adding certain predefined superclasses (mixins) to their class definitions. Typical desired behaviors are to move, select, or mark items. The algorithms ensuring a consistent image on the screen are provided by views. Views can also be declared as scrollable.

The system evaluates generic functions to draw and delete visible items if required. For the nodes in the class browser example (Figure 2) we might use the following definitions.

```
(defclass label-view-item
 (movable-view-item-mixin view-item)
 ((label :initarg :label :accessor label)))
(defun make-label (...) ...)
```

Class `label-view-item` is defined by inheriting the base class for view items and a 'mixin' which allows to move items interactively. The generic function `view-item-draw` is evaluated to draw view items:

```
(defmethod view-item-draw
 :after ((item label-view-item) view dialog)
 (let ((position (view-item-position item))
       (size (view-item-size item)))
  (frame-round-rect dialog std-gcontext ...)
  (move-to dialog ...)
  (draw-string dialog std-gcontext (label item))))
```

The drawing functions use *graphical contexts*[4] defining coordinate transformations and drawing attributes. The position and the drawing vector of a view item define a *drawing rectangle* (see Figures 4 and 5) which is used as clipping rectangle. Specialized methods depending on particular drawing contexts (e.g. views or dialogs) can easily be defined.

3.2 Referencing View Items

View items may reference other view items. References to other view items are also declaratively described. The corresponding coordinates are automatically computed. This scheme defines edges as view items which reference their corresponding nodes. For instance, the standard drawing function of an edge retrieves the references to its nodes and computes coordinates used to draw the connecting line. References are described by a so-called *reference box* (`:rbox`). A reference box consists of two view item, the

[2] The indentation of the boxes is used for demonstration purposes only.

[3] Layout forms are usually defined as macros evaluating the right expressions (in the right scope).

[4] The notion of a graphical context is defined by views.

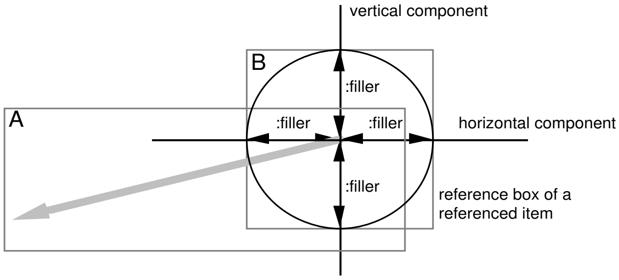

Figure 4: Referenced Location of Item B.

referencing item (e.g. A) and the referenced item (e.g. B), and a pair of horizontal and vertical coordinates specifying the location which is referenced.

```
(:rbox A B (:horizontal :filler :reference :filler)
           (:vertical :filler :reference :filler))
```

This box defines a reference from item A to item B. The keyword `:reference` specifies the referenced point. Figure 4 shows an example for this type of reference. The start point of item A (gray vector) is defined as reference to the center of item B (circle). The reference box of item B is shown as gray rectangle. It is defined as the drawing rectangle of item B. The usage of filler specifications in the horizontal and vertical coordinates ensure the centering of A's reference point to B.

Edges of graphs are also represented as view items. We apply the concepts introduced insofar to define a class `line-view-item` representing edges. The end points of an edge are specified as references to its nodes. The drawing function of `line-view-item` uses the predefined functions `references-of-this-item` and `reference-position` in order to compute the coordinates of the two points defining the edge (see Figure 5).

```
(defclass line-view-item (view-item) ())
(defmethod view-item-draw
 :after ((item line-view-item) view dialog)
 (let* ((references (references-of-this-item item))
        (p1 (reference-position (first references)))
        (p2 (reference-position (second references))))
  (move-to dialog p1)
  (line-to dialog std-gcontext p2)))
```

References may also be used to define an erase function specialized for edges. Only the line representing an edge is erased.

```
(defmethod view-item-undraw ((item line-view-item)
                              view dialog
                              position size references)
 (using-gcontext
  ((eraser-gcontext :pen-pattern *white-pattern*))
  (let ((p1 (reference-position (first references)))
        (p2 (reference-position (second references))))
   (move-to dialog p1)
   (line-to dialog eraser-gcontext p2)))))
```

If the location of the rounded rectangles (Class-1, Class-2) is changed the reference points P1 and P2 are recomputed and the line is redrawn.

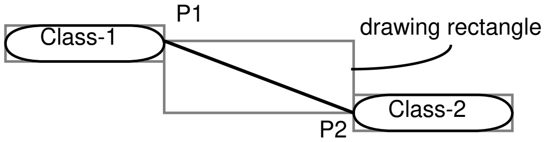

Figure 5: Location of edges defined by node references.

The layout specification of our class browser uses reference boxes to define node-connecting edges. The following specifications replace the DAG specification in Section 2.

```
(setf (layout graph-view) ...
 (:dag
   (list (find-class class-name)) ; list of DAG roots
   #'class-direct-subclasses ; successor function
   *hierarchy-depth* ; max. expansion depth
   #'(lambda (class) t) ; expansion predicate
   #'(lambda (class) ; node-creating function
      (make-label (class-name-as-string class)))
   #'make-line-view-item ; edge-creating function
   #'western-reference ; start point of edge
   #'eastern-reference)) ; end point of edge
(defun western-reference (referencing-object
                          referenced-object)
 "Definition of reference points with :rbox form"
 (:rbox referencing-object referenced-object
        (:vertical :filler :reference :filler)
        (:horizontal :reference :filler)))
(defun eastern-reference ...)
```

4 The Box Layout Algorithm Revisited

This section discusses the basic box layout algorithm in more detail. Important features of this algorithm are the allocation of arbitrary (interaction) objects conforming to the underlying protocol. The size of interaction objects may depend on predefined values or space constraints. These constraints can be expressed by minimal and maximal expansion values. Their usefulness is exemplified by another example. This example is part of a user interface of a CLOS inspector and browser which has been implemented within our framework. The inspector displays a window consisting of tabular subwindows. Figure 6 shows information about a class `tv:window`. It displays the class precedence list, super and subclasses, slot information, and direct (locally defined) methods. If necessary, tables may be scrolled provided the space available to a table is not sufficient for displaying all table elements.

The corresponding layout specification is composed of several parts and the constraints are partially computed at runtime. Therefore, we only explain its general outline. The upper rows are specified as horizontal boxes containing three frame boxes which represent tables. The width and height of the frame boxes are defined by constrained fillers. Each filler acts like a spring. All fillers compete for the space available to them which, in fact, is constrained by their surrounding horizontal box. Therefore, after a relaxation process each filler acquires 1/3 of the available space. Roughly, the horizontal boxes compete for the vertical space in a similar manner. Minimal and maximal

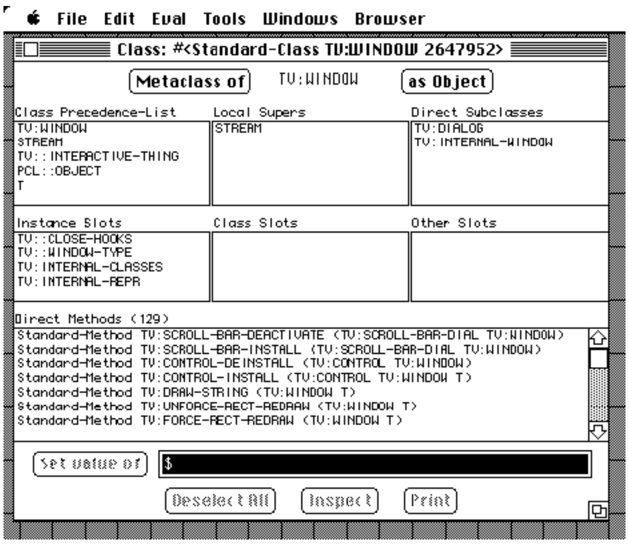

Figure 6: CLOS class inspector window.

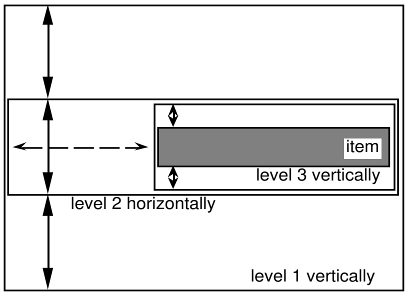

Figure 7: Schematic representation of nested boxes. Arrows represent fillers.

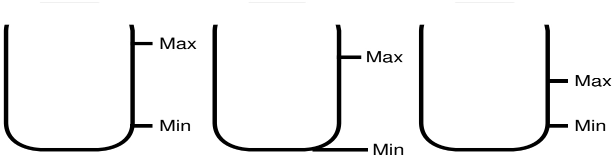

Figure 8: Water jugs modeling fillers.

constraints guarantee that sparse tables have a visually appealing uniform shape.

Our notion of constrained fillers is important for describing layout. Our formalism provides the user with a fast layout algorithm which is also easy to understand and anticipate. Boxes may be arbitrarily nested but only fillers at the same level compete for the available space. Figure 7 exemplifies this feature.

The (vertical) fillers of level 1 are independent of the (vertical) fillers of level 3. This restriction reduces considerably the computational complexity of the layout algorithm. Another advantage is that the semantics of constrained fillers are easy to comprehend by the user. Even this scheme requires a simple relaxation algorithm for satisfying these constraints. Figure 8 illustrates our algorithm.

Each water jug models a filler with its minimal and maximal expansion value. The current extension of a filler corresponds to the water contents of a jug. The space available to fillers is modeled by an external water reservoir with capacity R_0. The extension of a filler is computed by the following steps:

1. Initially each water jug j is filled to a level L_0^j. It is guaranteed that every water jug j is at least filled to its minimum $\min_j$. An additional demand for water which cannot be satisfied by the external water reservoir[5] is satisfied by a "water pipe". Then, the remainder R_1 of the water reservoir is distributed.
2. Each water jug gets a portion $P_i = R_i/N_i$ of the reservoir (be N_i the number of water jugs in the ith iteration). Step 1 may have caused for different water levels of the jugs. Therefore, every jug is filled to a common minimal level
$$L_i = M_i + P_i \text{ with } M_i = \min_{j \in 1..N_i} L_{i-1}^j$$
3. Caused by the minimum satisfaction guarantee the level of a jug may already be higher than L_i or its maximum $\max_j$. The amount of water of a jug whose level[6] exceeds either L_i or $\max_j$ is returned to the reservoir. A jug which has reached its maximal level becomes inactive ($N_i = N_i - 1$).
4. Start again with step 2 ($N_{i+1} = N_i$; $i = i + 1$) provided the water reservoir is not empty and there is at least one jug j left whose water level is below $\max_j$. This algorithm terminates if all jugs have become inactive ($N_i = 0$) or the water reservoir is empty.

The algorithm's termination is guaranteed since each cycle either makes one jug inactive or removes water from the reservoir. If no jug becomes inactive at least one jug is filled to the level M_i. Rounding errors are summed up. As final step this sum is rounded and added to the last filler.[7]

5 Related Work

The Symbolics™ programming environment Genera™ offers also means for specifying layout of windows. Subwindows (panes) can be arranged in a frame in columns or rows. Window sizes can be determined as absolute (fixed), relative, or with respect to the objects being allocated. These features can be compared with our box model. Filler specifications are also supported. Size specifications can be constrained by minimal and maximal distances. Genera only supports layouts for panes, but our layout algorithms can be applied to every object conforming to the

[5] This is an indication that the available space is not sufficient to fulfill the space requirements. Items could extend beyond the boundary of their surrounding box.

[6] The capacity of a jug be temporarily unlimited.

[7] This is needed for frame boxes which may otherwise not exactly fit to their box's lower border.

underlying abstract protocol. Genera offers the notion of presentation types which can be compared with our view item classes. Presentation types are also associated with handling of user input. In contrast to Genera our approach offers a more uniform and orthogonal layout scheme combined with a compact and elegant TeX-notation.

Recently, two other approaches have been proposed which also use TeX-like layout schemes for user interfaces. They also use constructs such as boxes and fillers for expressing window layout. The InterViews System [13] is a user-interface toolkit based on X windows and implemented in C^{++}. Fillers and boxes are implemented as objects. With respect to our Lisp environment we found the representation of boxes as a combination of ordinary lists and macros more efficient for manipulation and pattern matching. Our layout scheme is in several respects more powerful than InterViews' scheme. A filler-like size specification of boxes is not possible in InterViews. Important and useful notions such as a frame box which constrains the size of its box element or a general box which invokes user-defined parsers for layout specifications are not available.

The FormsVBT system [1] offers a two-view approach to designing user interfaces. The layout of a dialog window can be specified using both a TeX-like textual and a direct-manipulative graphical representation. Changes made in either representation are immediately updated in the other representation. FormsVBT is implemented in a dialect of Modula-2. Its specification language supports no macros and offers no support for new box types and layout schemes. Furthermore, we see the problem that the functionality of the textual notation has to conform with the graphical user interface. Mostly, this requires to reduce the functionality of the textual notation.

6 Summary and Future Work

This paper presented a compact, flexible notation for specifying layout of graphical objects. This notation is fully integrated into a Lisp environment based on CLOS. Advantages of this TeX-like declarative formalism are its expressiveness, user-predictable layouts, and efficient implementation schemes.

Next steps might be to combine the advantages of TeX-style notations with the general flexibility of constraint systems. Furthermore, research is necessary to extend this approach to 2-1/2 or 3-D layout.

Acknowledgements

We like to thank Ken Kahn for comments on a draft of this paper. The first author has been partly supported by a DAAD scholarship granted by the NATO science committee.

References

[1] G. Avrahami, K.P. Brooks, M.H. Brown, A Two-View Approach to Constructing User Interfaces, *ACM Computer Graphics* **23**, 3 (July 1989), 137–146.

[2] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon, Common Lisp Object System Specification, *ACM Sigplan Notices* **23**, 9 (Sept. 1988).

[3] A. Borning, The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems* **3** (1981), 353–387.

[4] M.H. Brown, Algorithm Animation, ACM Distinguished Dissertations Series, MIT Press, 1988.

[5] P.T. Cox, F.R. Giles, T. Pietrzykowski, Prograph: a step towards liberating programming from textual conditioning, In: Proceedings, *1989 IEEE Workshop on Visual Languages,* Rome (Italy), Oct. 4-6, IEEE Computer Society Press, 1989, pp. 150–156.

[6] M. Eisenstadt, M. Brayshaw, The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming, *Journal of Logical Programming* **5** (1988), 277–342.

[7] V. Haarslev, R. Möller, VIPEX: Visual Programming of Experimental Systems, In: *Visual Languages and Visual Programming,* S.K. Chang (ed.), Plenum Press, New York and London, 1990, pp. 185–212.

[8] V. Haarslev, R. Möller, A Framework for Visualizing Object-Oriented Systems, In: Proceedings, ECOOP/OOPSLA'90, *European Conference on Object-Oriented Programming* and *Object Oriented Programming: Systems, Languages and Applications,* Oct. 21-25, 1990, Ottawa/Canada, *ACM Sigplan Notices* **25**, 10 (Oct. 1990), 237–244.

[9] V. Haarslev, R. Möller, Visualization and Graphical Layout in Object-Oriented Systems, Technical Report, System Sciences Lab, Xerox PARC, 1990.

[10] K.M. Kahn, V.A. Saraswat, Complete Visualizations of Concurrent Programs and their Executions, In: Proceedings, *1990 IEEE Workshop on Visual Languages,* Skokie/IL, Oct. 4-6, 1990, IEEE Computer Society Press, 1990, pp. 7–14. .

[11] S. Keene, Object-Oriented Programming in CLOS - A Programmer's Guide to CLOS, Addison-Wesley, 1989.

[12] D.E. Knuth, TeX and Metafont – New Directions in Typesetting, Digital Press, 1979.

[13] M.A. Linton, J.M. Vlissides, P.R. Calder, Composing User Interfaces with InterViews, *IEEE Computer* **22**, 2 (1989), 8–22.

[14] J.H. Maloney, A. Borning, B.N. Freeman-Benson, Constraint Technology for User-Interface Construction in ThingLab II, *ACM Sigplan Notices* **24**, 10 (1989), 381–388.

[15] R. Möller, AI-Based Visualization Tools in Object-Oriented Systems (in German), *Technical Report* FBI-HH-B-149/90, University of Hamburg, Computer Science Department, 1990.

[16] P.A. Szekely, B.A. Myers, A User Interface Toolkit Based on Graphical Objects and Constraints, *ACM Sigplan Notices* **24**, 10 (1989), 36–45.