Visualization and Graphical Layout in Object-Oriented Systems[†]

Volker Haarslev[*] and Ralf Möller

University of Hamburg, Computer Science Department
Bodenstedtstr. 16, D-2000 Hamburg 50, F.R. Germany

haarslev@informatik.uni-hamburg.de
moeller@informatik.uni-hamburg.de

Phone: +49 (40) 4123-6137 or 6128 Fax: +49 (40) 4123-6530

Published in *Journal of Visual Languages and Computing* **3**, 1 (March 1992) 1–23.

[†] This paper is a major revision combining and extending two other papers [15, 16].

[*] This paper has mainly been written when the author was a visiting scientist at Xerox Palo Alto Research Center.

Abstract

This paper describes two aspects of visualizing program systems within the object-oriented paradigm: layout specifications for graphical objects and associations of visualization and application objects. The layout approach is based on a notation similar to the TEX text formatting language. It has been extended and generalized for specifying graphical layout of user interfaces and arbitrary objects. Our simplest scheme offers specifications similar to TEX's box-and-glue metaphor. Size and position of virtual boxes and glue can be specified by simple constraints. In the second part of the paper we show how the CLOS (Common Lisp Object System) meta-level architecture can be exploited to associate visualization and application objects. We show how several useful techniques such as indirect values, slot and method demons, and instance-specific meta-objects can be implemented using CLOS. These visualizations techniques require no source code modifications of application systems. We demonstrate the feasibility of our approach using application domains such as CLOS debugging and constraint systems.

Table of Contents

List of Figures

1 Introduction

Although programming has mostly been done in textual terms, users have always had a notion of visualizing their programs. Users entered programs as lines of text, but soon thereafter they began to use indentation and comments for separating or emphasizing particular program parts. They developed tools for pretty-printing and formatting source code. Moreover, modern programming environments offer debugging tools such as browsers and inspectors, which provide users with views of program structure and execution states. But these views display their information more textually than visually (pictorially). A further shortcoming of current environments is their lack of offering program designers adequate tools for visualizing and animating programs. We propose a classification scheme which distinguishes visualization by two kinds of strategies: structural and conceptual visualization.

Structural visualization uses program and data structures to generate relevant geometrical information for graphic substrates. An important problem related to this kind of visualization is that conceptual information about data can only indirectly be derived (e.g. 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 Nassi-Shneiderman diagrams. The Transparent Prolog Machine [10] is an example for relational or logic systems. A more radical approach is presented by Pictorial Janus [19]. It defines complete visualizations of concurrent logic programs and captures static as well as dynamic information about these programs. Furthermore, there exist many approaches to visualizing data flow in functional systems. One of the early systems was PICT/D [12]. Examples of newer systems are ConMan [17] and Prograph [7]. An overview of data-flow environments can be found elsewhere [18]. Regarding object-oriented systems a diagramming approach to tracing message passing [9] has been implemented as an extension of a Smalltalk-80 debugger. GraphTrace [23] is also intended for understanding behavior of objects. It provides graphical traces of program executions. All these approaches are primarily focused on the structure of computations. They offer only poor support for visualizing concepts of domains that are represented or modeled by programs. We use the notion of "conceptual visualizations" for advanced graphics reflecting domain concepts. *Conceptual visualizations* of programs are mostly hand-coded. 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.

In this contribution we discuss schemes for (aesthetically) laying out combinations of graphical objects. With respect to forms-oriented user interfaces, allocation of space and positioning of objects 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 (at least partially) addressed by many constraint-oriented systems. ThingLab I [4] and II [29] are examples for describing layout and form of graphical objects with constraints. Garnet [30] is also a toolkit using techniques such as constraints and active values.

In contrast to the constraint-oriented approaches mentioned above, we decided to provide a simpler but more compact and predictable notation for specifying layout. In the following we describe an approach to specifying layout of graphical objects, which is based upon TEX-like layout specifications. Users can define layout descriptions declaratively. However, layout specifications can also be computed by higher-level modules, e.g. using pattern matching techniques. Furthermore, our approach has the advantage of requiring only $2n + \log(n)$ steps, be n the number of layout elements. Thus, our algorithm has a computational complexity of $O(n)$.

The second topic discussed in this paper concerns the connection between application and visualization. We show how the CLOS Meta-Object Protocol might be used to design a component that links application to visualization objects and vice versa.

Throughout this paper we present several examples describing our layout notation and meta-level techniques. Since our implementation is based on a Macintosh Lisp environment, these examples are given in (simplified) Lisp code and assume a basic knowledge of Lisp and CLOS.

The remainder of this paper is structured as follows. Section 2 introduces our declarative layout specifications and demonstrates a CLOS class browser and inspector, which serve as examples for the flexibility of our approach. Section 3 introduces user-defined layout schemes and mechanisms for specifying references (local dependencies) between graphical objects. Section 4 explains our basic layout algorithm in more detail. The second part of our paper begins with Section 5 and discusses how to use the CLOS meta-object protocol for program visualization. It demonstrates some of these considerations by using a simple constraint system as example domain. Section 6 summarizes the current status of our implementation. Section 7 compares our approach with related work concerning layout descriptions. This paper concludes with a summary and a discussion of future work.

2 Layout Specifications

One important requirement in the design of user interfaces is the ability for specifying the layout of dialog windows. These windows usually consist of several elements implemented as graphical objects. Therefore, window layout can be expressed as spatial relationships between window elements. Within our framework layout is specified by an abstract notation. The underlying concept of this notation is based on the "box-and-glue" metaphor of the TEX text formatting language [24].

Boxes describe layout of rectangular regions of screen space. They have a type and are hierarchically organized. Boxes consist of window elements, (variable) white space (referred to as *fillers*), and other boxes. It is important to mention that our system implements layout boxes and fillers as *virtual objects*, i.e. not as special windows. Having virtual layout objects is advantageous in window systems where no lightweight windows are directly supported (e.g. on a Macintosh). Strategies for laying out and positioning box elements are defined by various box types. Our system offers a set of predefined box types implementing basic layout algorithms. Box types supporting more general layout algorithms are discussed in Section 3.

2.1 Fundamental Layout Boxes

Our basic building blocks for layout specifications are *horizontal* and *vertical boxes.* Elements of these boxes are laid out horizontally and vertically, respectively. A box specification consists of a keyword indicating the box type, a size specification which might be empty, and a list of box elements. A horizontal or vertical box (`<box-type>` either `:hbox` or `:vbox`) is specified by the Lisp form `(<box-type> (:width h :height v) box-item-1 ...)`. The size of a box can be specified as a fixed or variable distance (filler) in either direction. In case of an empty size specification, the width and height of a box are set to a filler with default constraints (see below) and determined by its surrounding box. If elements require more space than available to their surrounding box, they are allowed to extend beyond the boundaries of their box. Boxes may also overlap one another.

It is also possible to define the 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. Thus, the box has a minimal size satisfying all lower bound constraints.

In general the layout algorithm keeps the size of box elements unchanged. But this behavior is not always desired. Therefore, we introduced a *frame-box* (`:fbox`) which constrains (i.e. modifies) the size of its element in order to match exactly the size of the frame-box. A frame-box may contain only one box element: `(:fbox (:width h :height v) box-item)`.

2.2 Filler Specification

Fillers specify distances in either horizontal or vertical direction. They may be used as box elements denoting white space and gluing together other elements. They are also used in size specifications of boxes.

Fillers can be specified as *fixed* (e.g. size 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. Their default constraints are zero as lower and box size as upper bound. 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.

The complete form specifying a constrained 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>)`.

2.3 Two Examples: CLOS Class Inspector and Browser

The first example demonstrates a simple dialog window. The three basic box types defined above are combined and applied to specify the layout of a window. This window (see Figure 1) shows a simple CLOS class browser displaying a class hierarchy. The right table contains all direct subclasses of the class listed in the left table. The contents of the tables can be replaced by direct super- or subclasses, i.e. they can be shifted (and scrolled) to focus on

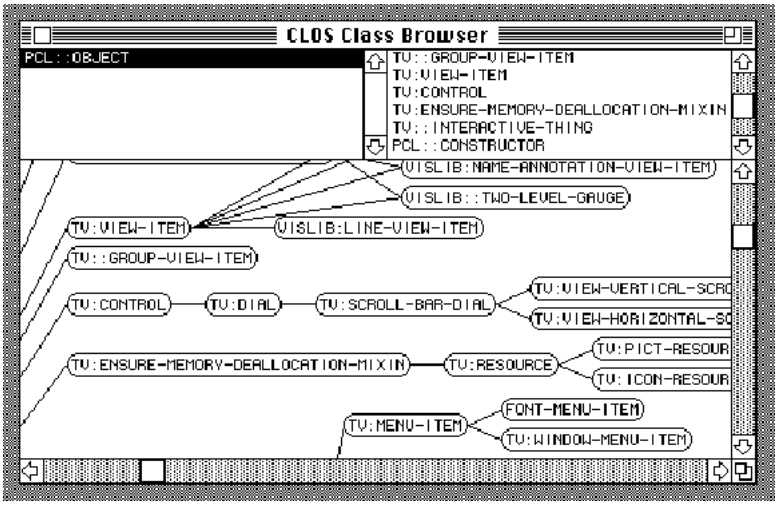

Figure 1: A DAG graph of a standard class hierarchy.

"interesting" classes. The lower part of the window displays a graph of the selected class hierarchy. The dialog window results from the following (schematic) layout specification (given as Lisp code). The layout facility is implemented by a class `layout-mixin` that provides methods for interpreting an initial layout description and updating a layout accordingly when the size of a corresponding dialog (or view) has been changed.

```
(defclass graph-view-scroller
  (layout-mixin scroller) ; inheritance of scroller makes a display area scrollable
  ()
  (:documentation "Combines scrollable views and the layout facility"))

(defclass class-browser-dialog
  (layout-mixin dialog)
  ()
  (:documentation "Defines dialog windows with layout"))

(let ((left-table (make-instance 'sequence-dialog-item ...))
      (right-table (make-instance 'sequence-dialog-item ...))
      (graph-view (make-instance 'graph-view-scroller ...)))
  (make-instance 'class-browser-dialog
    :layout
    (:vbox (:width :filler :height :filler) ; main vertical box
      (:hbox (:height 1/4 :width :filler)   ; top row
        (:fbox () left-table)
        (:fbox () right-table))
      (:fbox () graph-view)))               ; bottom row
  (setf (layout graph-view) ...))
```

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 horizontal

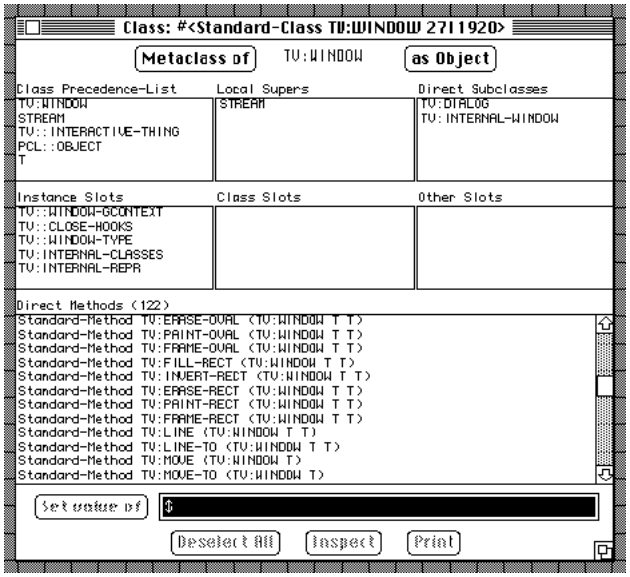

Figure 2: CLOS class inspector window (default size).

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). The second item of the vertical box is a frame-box surrounding the box element `graph-view` which generates the class graph. A layout form which is similar to that defining `graph-view` is shown in Figure 4.

The second example is part of a user interface of a CLOS inspector. It demonstrates the use of fillers with minimum and maximum length specifications. The inspector displays a window consisting of tabular subwindows. Figure 2 shows information about a class `window`. It displays the class precedence list, direct 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, 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. All fillers compete for the horizontal space available to them, which, in fact, is constrained by their surrounding horizontal box. Therefore, after a relaxation process each filler acquires one third of the available space. Roughly, the horizontal boxes compete for the vertical space in a similar manner. Minimal and maximal constraints guarantee that sparse tables have a visually appealing uniform shape.

Our notion of constrained fillers is important for describing a flexible window layout. For instance, Figure 3 shows the same window as in Figure 2 except that the vertical and horizontal space available to the window has been enlarged. The table displaying the direct methods of

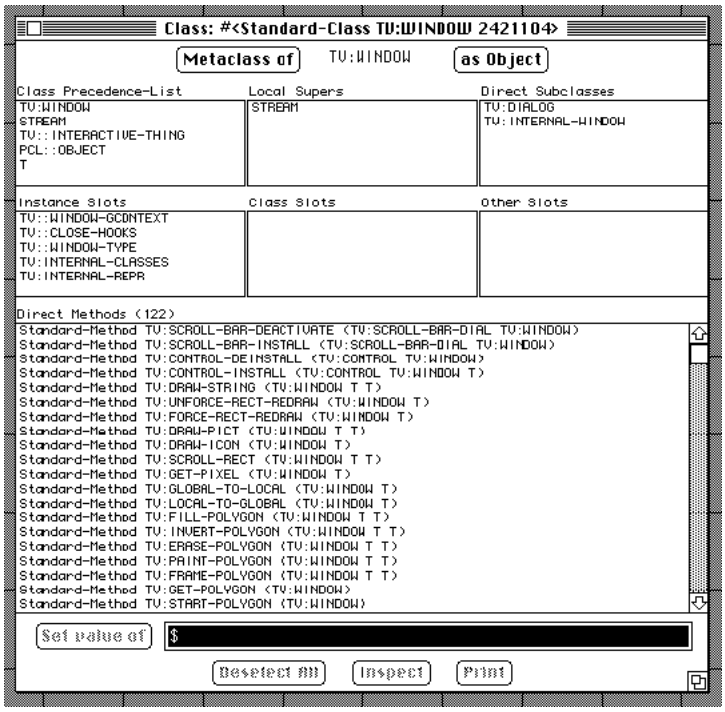

Figure 3: CLOS class inspector window (vertically and horizontally enlarged).

class `window` adapted to the new space constraint. The additional vertical space was completely consumed by this table since it is the only element with a filler constraint whose upper bound has not yet been reached.[1]

2.4 Local Variables in Layout Forms

Our current layout scheme provides no support for specifying dependencies between box items which are not elements of the same box. For instance, the height of box elements cannot depend on the width of arbitrary other elements. This restriction could be relaxed by introducing *local variables* in layout forms. These variables could represent box attributes such as the actual width and height of a box. The variables could serve as constants within the scope of a form. The following example shows a useful application of this feature.

```
(let ((item-1 (make-dialog-item ...))    ; create two window elements
      (item-2 (make-dialog-item ...)))
  (:vbox (:width (:filler :bind ?total-width)) ; ?total-width is local variable
    :filler ; vertical centering
    (:hbox ()
      20 ; left border (in pixel)
```

[1] Actually, its upper bound for vertical space is set to a height which would allow to display the whole list of direct methods at once provided that sufficient screen space is available.

```
    (:fbox (:height (truncate (/ ?total-width 2))) item-1) ; use local variable
    :filler ; align frame-boxes to left and right
    (:fbox (:height (truncate (/ ?total-width 2))) item-2) ; use local variable
    20) ; right border (in pixel)
  :filler)) ; vertical centering
```

One interpretation of this specification might be as follows. The evaluation of the form `:bind ?total-width` depends on its lexical context. In this case the value of `?total-width` is set to the actual width of the vertical box, which is constrained by the width of the surrounding window. Then, this value is used to determine the height of `item-1` and `item-2`. Therefore, the height of `item-1` and `item-2` depends on the total width of their outermost vertical box.

3 User-Defined Layout Schemes

The previous section introduced our three basic box types and their associated layout algorithms. Apparently, it is not reasonable to describe every layout with the box-and-glue metaphor. An obvious example is a set of nodes arranged as a graph. We provide a means for integrating these special layout descriptions with the box-and-glue mechanism. Our layout language has the notion of a *user-defined box*: `(<layout-name> <arg-1>...<arg-n>)`. For instance, with `<layout-name>` being `:dag` we have `(:dag <roots> <successors> <depth> <appearance>)`, i.e. the layout of directed acyclic graphs (DAGs) is specified (see Figure 4). In this (simplified) example the items to be arranged are defined inductively by a set of roots, a function responsible for generating successors of a node, a maximal graph depth, and a function which is used to compute the graphical representation of nodes.

Position and size of the associated box are defined implicitly by a closure rectangle around all graph items (see Figure 4). 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 user-defined box. For instance, a new layout interpreter can be provided by the following form.[2]

```
(deflayout :dag (layout-specs)
  "Returns (generated and) laid out DAG elements."
  (interpret-dag-layout layout-specs))
```

The layout name (e.g. `:dag`) of a user-defined box form is used as a key to discriminate the corresponding layout interpreter, the rest of the form is passed to the parameter `layout-specs`. In order to determine size and position of objects found in `layout-specs` certain generic functions (e.g. `(setf box-item-size)`, `(setf box-item-position)`) have to be used. Appropriate methods may be supplied for items to be positioned using the layout descriptions.

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

```
(let ((upper-offset 10)
      (left-offset 10))
  (:vbox ()
    upper-offset
    (:hbox () left-offset
      (:dag *roots*
            #'successors
            *max-depth*
            #'appearance
            ...))))
```

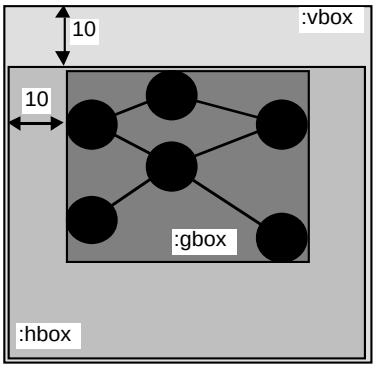

Figure 4: A general layout specification in combination with a box-style layout.[3]

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

3.1 Local Relations Between Graphical Objects

Using the layout descriptions described in the previous sections, it is easy to determine the position of graphical objects relative to a global window, say. In other domains (such as graph layout) objects are dependent on one another (e.g. concerning size and position). In our graph example edges are dependent on graph nodes, i.e. position (and size) of the edges can be computed by referring to the nodes. Since nodes can be moved, the referenced position may change. We generalized these ideas in the following way.

References can be established between any two graphical objects. In technical terms: the reference facility can be supplied to graphical objects by defining `reference-mixin` as one of their superclasses. Corresponding coordinates (stored in reference objects) are automatically (re)computed and appropriate redisplay methods are evaluated to update the display accordingly when a referenced object is moved (or dragged). The drawing function of an edge retrieves the computed references and draws the connecting line accordingly.

References are specified by a *reference box* (`:rbox` ...). A reference box consists of two view items, the 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` is used to specify the horizontal and vertical coordinates of `A`'s reference point inside of `B`'s drawing rectangle. Fillers ensure the centering of this reference point. Figure 5 illustrates this type of reference. The start point of the gray vector defines `A`'s reference to the center of `B` (circle). The orientation of the vector depends on `A`'s position which is not shown in this example. The reference box of `B` (its drawing rectangle) is shown as gray rectangle. The next example

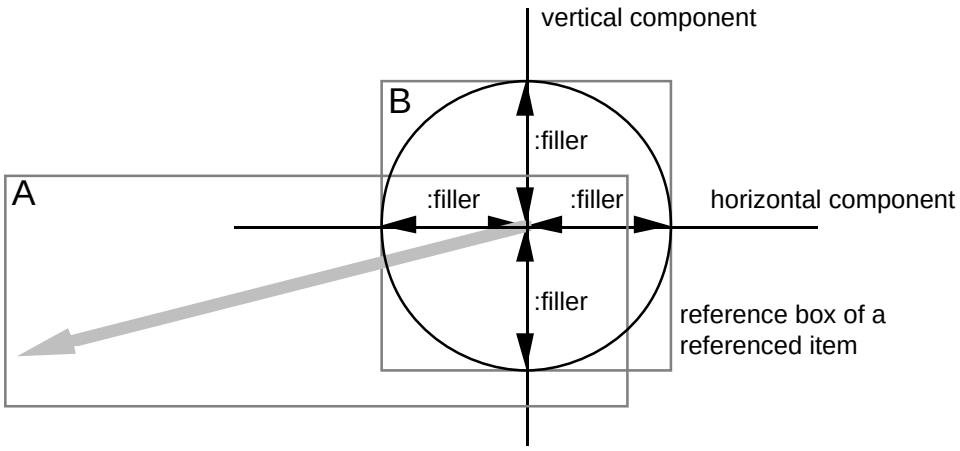

Figure 5: Location referenced inside of item B.

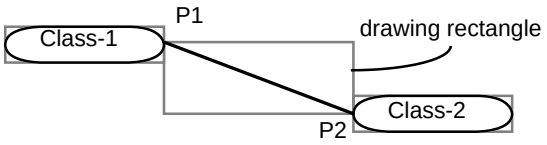

Figure 6: Location of edges defined by node references.

defines a reference point whose location is 3 pixel above to and left from the lower right corner of item B.

```
(:rbox A B
       (:horizontal :filler :reference 3)
       (:vertical :filler :reference 3))
```

We apply the concepts introduced insofar to define a class `edge-view` representing edges in a graph. The end points of an edge are specified as references to corresponding nodes. In order to compute the coordinates of the two points defining the edge (see Figure 6), the drawing function of `edge-view` uses the predefined functions `references-of-this-view` and `reference-position` .

```
(defclass edge-view
  (reference-mixin view) ; MCL class view provides a local coordinate system
  ()
  (:documentation "Edge in a graph."))

(defun make-edge (...) ...)

(defmethod view-draw-contents ((view edge-view))
  (let* ((references (references-of-this-view view))
         (p1 (reference-position (first references)))   ; first point
         (p2 (reference-position (second references)))) ; second point
    (move-to view p1)                    ; set pen position
```

```
    (line-to view standard-gcontext p2))) ; draw edge between p1 and 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.

The layout specification of our class browser uses reference boxes to define node-connecting edges. The following specifications replace and supplement the specification given in Section 2.3.

```
(setf (layout graph-view)
      (:vbox ()
        10     ; 10 Pixel upper border
        (:hbox ()
          10   ; 10 Pixel left border
          (: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-node (class-name-as-string class)))
                #'make-edge                      ; 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. A function is used
   to handle the parameter bindings."
  (:rbox referencing-object referenced-object
         (:vertical :filler :reference :filler)
         (:horizontal :reference :filler)))

(defun eastern-reference ...)
```

3.2 Named References

The reference elements introduced insofar are implemented as a set-like data structure. For instance, it is not possible to define directed references since their start and end points could not be identified. Therefore, we extended our reference boxes for supporting a naming mechanism. Names may be assigned to descriptions of reference boxes. These names enable the user to define higher-level abstractions such as directed references. For instance, the following specifications define the start and end point of an object `arrow`, which points from the lower right corner of object `node-1` to the upper left corner of object `node-2`.

```
(:rbox :arrow-start arrow node-1
       (:horizontal :filler :reference)
       (:vertical :filler :reference))

(:rbox :arrow-end arrow node-2
```

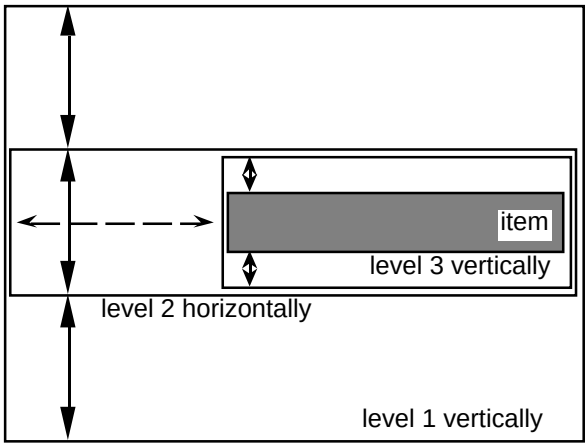

Figure 7: Schematic representation of nested boxes. Arrows represent fillers.

```
(:horizontal :reference :filler)
(:vertical :reference :filler))
```

The predefined function `find-reference` can be applied to a reference name (e.g. `:arrow-start`) and a list of references and returns the corresponding reference. For instance, this feature is used by an arrow-drawing method for determining the arrow direction.

```
(defmethod view-draw-contents ((view edge-view))
  (let* ((references (references-of-this-view item))
         (p1 (reference-position (find-reference ':arrow-start references)))
         (p2 (reference-position (find-reference ':arrow-end references))))
    (draw-arrow view standard-gcontext p1 p2)))
```

These examples demonstrated the usefulness of our layout extensions, which have been fully integrated with our box layouts. We think of reference boxes as a convenient, simple, and computationally cheap way for specifying local dependencies between graph elements. Since efficiency is a major concern of our approach, the next section explains some of our considerations regarding our box layout algorithm in more detail.

4 The Box Layout Algorithm Revisited

The important features of our basic box layout algorithm have already been explained in the previous sections. This formalism provides the user with a fast and flexible layout algorithm, which is also easy to understand and anticipate. In order to keep this formalism simple, we decided to restrict the dependencies between boxes and fillers, i.e. their mutual influence on size and position of graphical objects. Boxes may be arbitrarily nested, but only fillers on the same box level compete for the available space. Figure 7 exemplifies this feature.

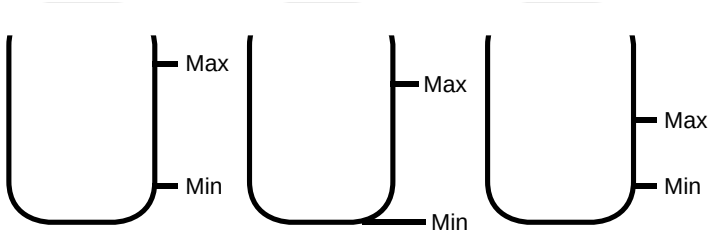

Figure 8: Water jugs modeling fillers.

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 constraint satisfaction. Figure 8 illustrates our algorithm.

Each *water jug* models a filler having a minimal and maximal expansion value. The current extension of a filler corresponds to the water level of a jug. The total 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 $\min_j$. In case of an additional demand for water, which cannot be supplied by an empty water reservoir,[4] this demand is satisfied by a "water pipe". If there is any water R_1 in the reservoir left, the remainder of this water is distributed according to the following steps.

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

$$L_i = M_i + P_i, \qquad \text{with } M_i = \min_{j \in 1..N_i} L_{i-1}^j$$

3. Caused by the minimum satisfaction guarantee, the level[5] of a jug j may already be higher than L_i or its maximum $\max_j$. The amount of water exceeding 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 when all jugs are inactive ($N_i = 0$) or the water reservoir is empty.

[4] 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.

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

The termination of this algorithm 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. The computation of the relaxation process uses integer arithmetic. Therefore, rounding errors are summed up and as final step this sum is rounded and added to the last filler.[6]

5 Meta-Level Techniques for Separating Application and Visualization

In order to ensure clear program structures, application and visualization objects should be separated. The second part of this paper discusses how to use the meta-object protocol of CLOS (see also elsewhere [21, 20]) for linking application and visualization layers. This link is mostly based on interesting events generated by application objects. These events have to be visualized in one way or another. The recognition of interesting events is a well-known problem since these events are very often only indirectly reflected by algorithms. We refer to Brown [5] for a detailed discussion of these problems.

Our approach associates visualization objects with given application objects without requiring any source code modifications to the application. We support multiple views as well as controllers for manipulating the application's data structures. Several other mechanisms have also been developed (Model-View-Controller-Scheme [13], CLUE [22], Presentation-Types [32, 6]).

As example application we chose a simple constraint net. There is no need to present the application code since everything can be found in detail elsewhere [34]. The application provides a simple model of a stock exchange scenario. When are some stocks to split? The participants have uncertain knowledge and are influenced by one another. A constraint net models these influences by propagating certainty-estimation intervals between 0 and 1. This interval of a 'broker' might be visualized by a gauge [34]. The implementation distinguishes assertion objects (brokers, mystics, virtual intermediates, etc.) and constraint objects (or, and). Figure 9 shows a snapshot of a program animation. The example configuration consists of gauges for assertions and simple nodes for constraints.

The visualization in Figure 9 can be built with the techniques described in the previous sections. The graph is defined by a set of participants and a successor function `stock-exchange-wizard`.

```
(defmethod stock-exchange-wizard ((participant assertion))
  "Wizard's information about connections of assertion objects."
  (assertion-constraints participant)) ; OR nodes

(defmethod stock-exchange-wizard ((participant constraint))
  "Wizard's information about connections of constraint objects."
  (list (constraint-output participant))) ; brokers, mystics, etc.
```

Another function, `application-visualization-coupler`, is used to define a mapping from

[6] This is needed for frame-boxes, which may otherwise not exactly fit to the lower border of their box.

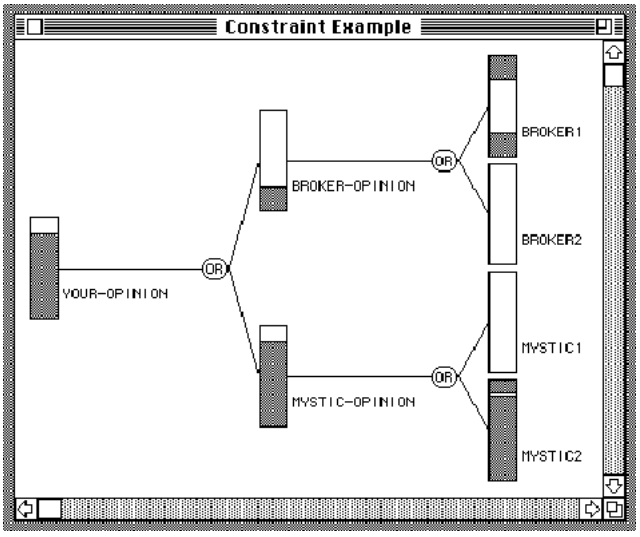

Figure 9: The upper and lower bounds are indicated by shaded rectangles [34]. The gauges show the estimation interval from 0 (bottom) to 1 (top).

application to visualization objects (see Section 5.2). Both functions are generic, i.e. different mappings may be specified for different classes of application objects.

5.1 Indirect Values for Visualization Objects

Visualization objects have to refer to objects of the application side. There should exist a "dynamic" binding which could be easily maintained provided that classes of visualization objects offer support for some kind of active values [2]. We present a simplified CLOS metaclass supporting non-nested active values which we call *indirect values.* Indirect values are defined by the form `(make-indirect-object object reader writer)` where `writer` is optional. The following definitions sketch an implementation using a new metaclass and a corresponding meta-level method for the generic slot accessor function `slot-value-using-class`. Writing to slots with indirect values can be implemented analogously.

```
(defclass indirect-slots-class (standard-class)
  ()
  (:documentation "Metaclass supporting indirect slot values."))

(defmethod check-super-metaclass-compatibility ((x indirect-slots-class)
                                                (y standard-class))
  t) ; We do not care about that in this paper (see Graube [14])

(defmethod slot-value-using-class ((class indirect-slots-class)
                                   object slot-descriptor)
  (let ((direct-slot-value (call-next-method)))     ; get slot value
    (if (indirectp direct-slot-value)               ; test for indirect value
```

```
    (funcall (indirect-reader direct-slot-value)  ; notify application object
             (indirect-object direct-slot-value))
    direct-slot-value)))
```

Visualization objects may have `indirect-slots-class` as their metaclass. With indirect slot values every slot access is delegated to the corresponding application object if required. Using the meta-object protocol it would be easy to determine all indirect objects or that indirect object referred to by a specific slot. The gauges for the stock exchange example use this metaclass to refer to the exchange participants. But what about the other direction: the gauges have to be "informed" when the participants' estimations of stock splits change.

5.2 Slot Demons for Application Objects

An assertion object has one slot for the lower and one for the upper bound estimation. The corresponding visualization objects have to be informed when either of these slot values change. The most obvious way to achieve this is to define the assertion class with a metaclass that allows *demon functions* to be attached to slots. The "real" value of a slot is a structure that provides a value facet and an `if-modified` facet [31]. All slot demon functions are evaluated when the slot value changes. The implementation of slot demons is similar to the one of indirect slot values. We introduce a metaclass `demon-slots-class` and define modified versions of `slot-value-using-class` and `(setf slot-value-using-class)`, which access the value facet. The latter one evaluates the demons in the `if-modified` facet. Slot demons should be made removable.

Thus, the function `application-visualization-coupler` mentioned above can be defined as follows.

```
(defmethod application-visualization-coupler ((participant assertion))
  (let ((assertion-gauge (make-two-level-gauge   ; with indirect values
                          (make-indirect-object participant
                                                assertion-lower-bound)
                          (make-indirect-object participant
                                                assertion-upper-bound))))
    (add-slot-if-modified-demon
      participant  ; object
      'lower-bound ; slot name
      #'(lambda    ; demon function
          (assertion-obj name-of-modified-slot old-value new-value)
          (gauge-update assertion-gauge)))
    (add-slot-if-modified-demon
      participant  ; object
      'upper-bound ; slot name
      #'(lambda    ; demon function
          (assertion-obj name-of-modified-slot old-value new-value)
          (gauge-update assertion-gauge)))
    assertion-gauge))
```

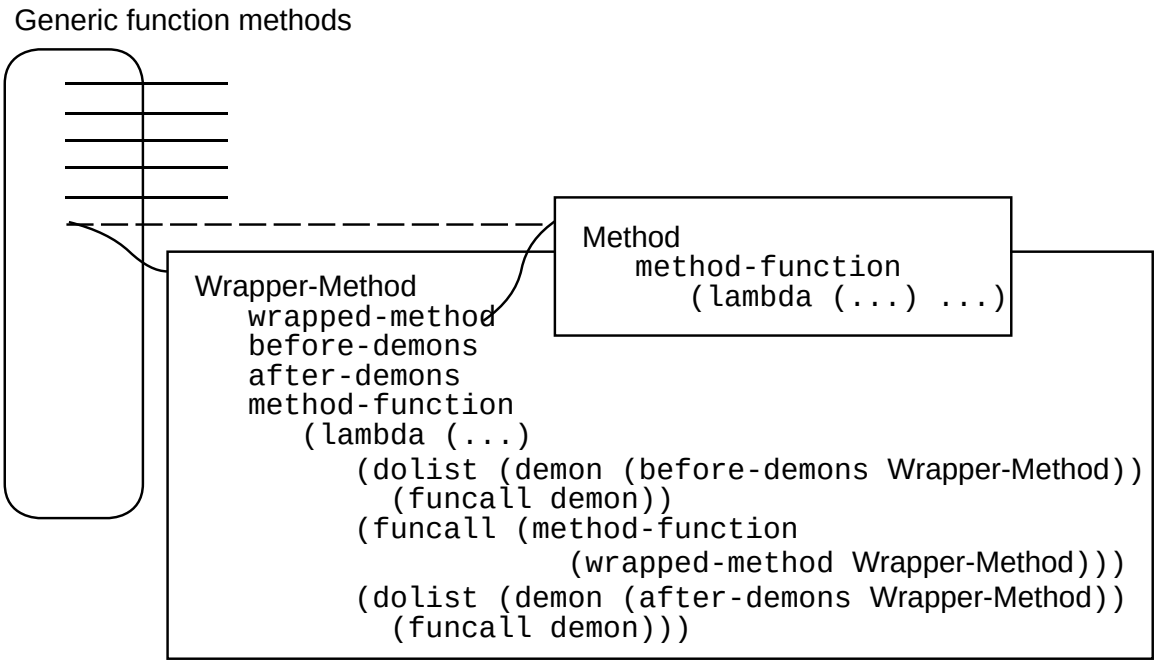

Figure 10: Outline of a wrapper method.

```
(defmethod application-visualization-coupler ((participant or-box))
  (make-node "OR"))
```

Demon functions are closures which provide access to the corresponding visualization object. The gauges for assertion objects (broker, etc.) use indirect values to access assertion objects. The objects representing constraints as nodes are the same as in the class browser example.

5.3 Method Demons

Slot demons offer an elegant way of defining slot accesses as interesting events and hence updating corresponding visualization objects. Not only slot accesses are subject to updating a visualization. Every method might define an event of interest. Slot accesses are only special cases. General *method demons* can be implemented using the meta-object protocol of CLOS. The idea is to wrap a method with a so-called *wrapper method* which has slots to refer to both the demon functions and the original method (see Figure 10). When all demons are removed the wrapper method itself is removed, too. In this case there is no overhead as with a metaclass which provides own methods for slots accesses (e.g. for indirect values) that overwrite the standard slot accessor methods.

A major disadvantage of this wrapping slot accessor is that demons are evaluated for all instances, i.e. they are slot but not instance-specific. Method demons do not solve the problem

of compound slot accesses, either. In the following two subsections we propose a solution to these problems.

5.4 Instance-Specific Meta-Objects

The CLOS meta-object system assigns to metaclasses the responsibility for both structure (implementation) and behavior of instances. There are other meta-level systems which distinguish between structural and computational meta-objects [11]. In this section we present ideas to provide some kind of dynamic meta-level influence in CLOS [8]. We implement meta-objects as instances of a class `standard-meta-object` which serves not as a CLOS metaclass. The standard slot access protocol, which uses the method `slot-value-using-class`, is analogously extended for these "simple" meta-objects.

```
(defclass standard-meta-object ()
  ()
  (:documentation
   "Class of all meta-objects that provide instance-specific meta behavior."))

(defmethod slot-value-using-meta-object ((mobj standard-meta-object)
                                         object slot-descriptor)
  (call-next-meta-method))

(defmethod (setf slot-value-using-meta-object) ((mobj standard-meta-object)
                                                object slot-descriptor)
  (call-next-meta-method))
```

Meta-objects can be assigned to instances with metaclass `extensible-standard-class`. This metaclass describes classes with instances that have one additional or implicit slot called `meta-objects` (see Figure 11). A set of meta-objects can be assigned to this slot.

An example method handling slot accesses is defined as follows.

```
(defmethod slot-value-using-class ((class extensible-standard-class)
                                   object slot-descriptor)
  (let ((slot-name (slotd-name slot-descriptor)))
    (if (eq slot-name 'meta-objects) ; prevent recursive slot access
      (if (slot-boundp object 'meta-objects)
        (call-next-method)
        nil)
      (let ((*meta-objects* (slot-value object 'meta-objects))
            (*meta-class-generic-function* #'(lambda () (call-next-method)))
            (*meta-object-generic-function*
              #'(lambda (meta-object)
                  (slot-value-using-meta-object meta-object
                                                object
                                                slot-descriptor))))
       (declare (special *meta-objects*
                         *meta-object-generic-function*
```

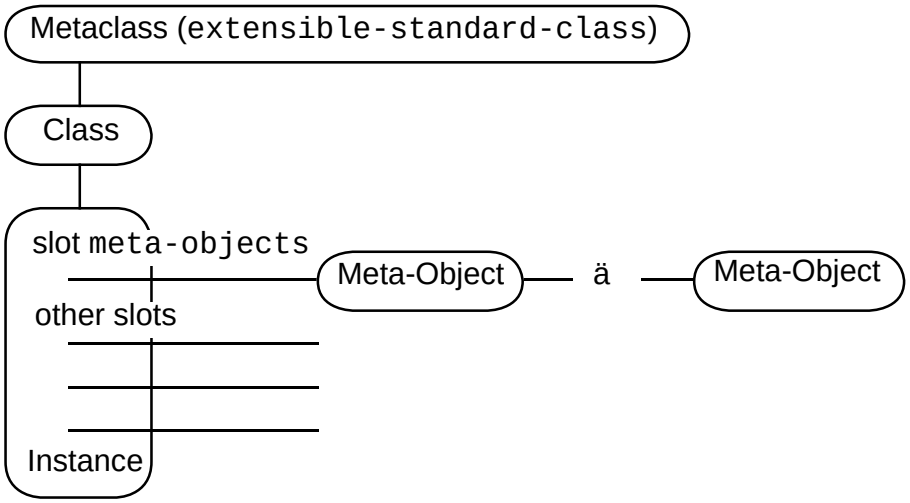

Figure 11: Meta-objects for instances with metaclass `extensible-standard-class`.

```
                    *meta-class-generic-function*))
  (if (null *meta-objects*)
    (call-next-method)
    (call-next-meta-method))))))
```

The function `call-next-meta-method` is comparable to the function `call-next-method`. It evaluates `slot-value-using-meta-object` for the "next" meta-object in the list of meta-objects (see Figure 11). The default behavior of `slot-value-using-meta-object` is to evaluate `call-next-meta-method` again (s.a.). This default behavior may be augmented or overwritten by subclasses of `standard-meta-object` (s.b.). When there are no meta-objects (left), `call-next-meta-method` invokes the "normal" slot access functionality of `standard-class`. There is some additional code needed to enable passing of different parameters to the next metamethod just as with `call-next-method`.

```
(defun call-next-meta-method ()
  (declare (special *meta-objects*
                    *meta-object-generic-function*
                    *meta-class-generic-function*))
  (if (endp *meta-objects*)
    (funcall *meta-class-generic-function*)
    (funcall *meta-object-generic-function* (pop *meta-objects*))))
```

We use these meta-level techniques to extend our constraint example. Using the protocol described above visualizations of particular instances can be provided with little programming effort. For instance, a meta-object could be defined by a class `visualizer-meta-object`. This class combines a visualization object with a list of interesting slots. Every writing access to these slots is followed by calling the instance-specific visualization object.

```
(defclass visualizer-meta-object (standard-meta-object)
  ((visualizer :initarg :visualizer
               :accessor visualizer
               :initform #'(lambda (&rest ignore) nil))
  (interesting-slots :initarg :interesting-slots
                     :reader interesting-slots))
  (:default-initargs :interesting-slots nil))

(defmethod (setf slot-value-using-meta-object) :after (new-value
                                                        (mobj visualizer-meta-object)
                                                        object slot-descriptor)
  (if (member slot-name (interesting-slots mobj))
    (funcall (visualizer mobj) object (slotd-name slot-descriptor))))
```

Be `your-opinion` the assertion object of our constraint example (see Figure 9). We add only to this object a corresponding meta-object which prints `your-opinion`'s decision about buying stocks. This behavior can be easily reverted by removing this meta-object from the implicit slot `meta-objects`.

```
(add-meta-object your-opinion
                 (make-instance 'visualizer-meta-object
                                :interesting-slots '(lower-bound upper-bound)
                                :visualizer
                                  #'(lambda (assertion slot-name)
                                      (if (> (assertion-lower-bound assertion) 0.75)
                                        (print 'buy) ; or any other visual feedback
                                        (print 'donot-buy)))))
```

Another behavior might be to temporarily modify a reading access to a slot value. After adding a meta-object of class `buying-indicator-meta-object` to the object `your-opinion`, each reading access to the slot `lower-bound` of `your-opinion` returns the slot value and a buying indicator.

```
(defclass buying-indicator-meta-object (standard-meta-object)
  ())

(defmethod slot-value-using-meta-object ((mobj buying-indicator-meta-object)
                                         object slot-name)
  (if (eq slot-name 'lower-bound)
    (let ((slot-value (call-next-meta-method)))
      (if (> slot-value 0.75)
        (values slot-value 'buy)
        (values slot-value 'donot-buy)))
    (call-next-meta-method)))

(add-meta-object your-opinion (make-instance 'buying-indicator-meta-object))
```

Instance-specific meta-objects have also been proposed by Maes [27, 28]. The difference to our approach is that (at least basically) only one meta-object may be assigned to an object

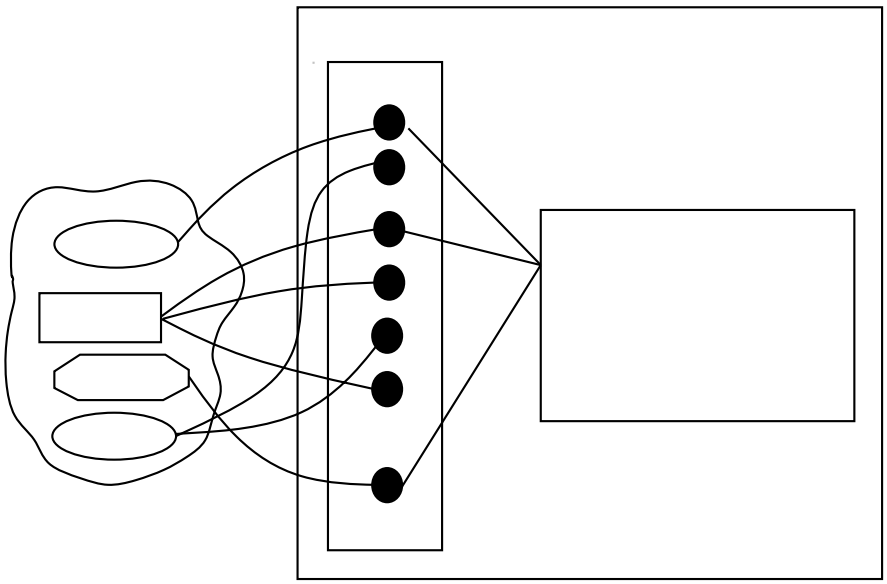

Figure 12: Instances `i-1` to `i-4` with slots `s-1` to `s-3` report new values to an event manager. The agenda is extended at the bottom.

at a certain time. One may of course argue that our CLOS implementation is a little impure because of using different mechanisms: metaclasses and meta-objects. Moreover, not all meta-objects may be compatible (see Graube [14] for a discussion). There also remains some overhead even when no meta-objects are attached to an instance.

5.5 Compound Events

The events of interest mentioned in the previous sections are only defined implicitly, i.e. there are no objects generated to describe events. However, an explicit representation of event conditions is needed to handle more complicated (compound) events. For instance, the situation in our stock example "all brokers have a lower bound estimation greater than 0.75" might be defined as a compound event. Then, a coordination problem arises since several objects are now concerned. It should even be possible to combine several events (e.g. value changes of different slots in different objects) as a compound event.

We need a declarative specification of such event conditions and a management system responsible for collecting and monitoring announcements of subevents rather than simple demons handling slot accesses directly (see Figure 12).

Subevents provide a data structure (in Figure 12 instances of these structures are represented by black dots) for storing event information (e.g. an object, its changed slot, the previous value, and the new value). The management system uses an agenda to access incoming subevents. The supervisor decides when agenda entries can be compiled to a compound event.

The main problem with compound events is the necessity to define the notion of a "step."

The definition of a step depends on an "interpretation" of the object system, e.g. different step interpretations might exist.

Meta-level programming can be used to define a certain step interpretation for an existing object system. We distinguish two kinds of steps: object steps and system steps. Let's have a look at the constraint example again. We assume that accesses to the slots `lower-bound` and `upper-bound` are recorded on the agenda. An entry of the agenda comprises the object together with the corresponding slot values of `lower-bound` and `upper-bound`.

An object step is induced when there already exists a slot access entry for a certain slot and the meta-system attempts to add another entry for this slot. If there is an entry for the other slot, the system combines the entries to an object step using the values of the current entry. If there is no such slot entry for the "other" slot, the current slot values of the object are used to determine its value. Afterwards, all used agenda entries of the corresponding object are deleted. The new entry is inserted.

System steps are defined analogously, just one level higher. A system steps consists of a list of objects with a sequence of slot values associated to them. As an example we consider two brokers as a (trivial) object "system." A system step is induced when there already exists an object step for one of the brokers and an attempt is made to create another object step just for the same broker object. If no step for the other broker has been recorded, its current slot values are copied and used for the system step. System steps, with the slot values recorded for each object, can be supplied to a visualization component for further processing.

We emphasize that this is only one possible proposal for an event manager, though it is suitable for our stock exchange example.

Other visualization systems (e.g. see in Linden [25]) record the whole agenda (or the stream of events) on a file so that a postmortem visualization can be provided (tractable only for 'small' systems).

6 Status of Implementation

The layout system and the meta-level techniques have been originally implemented using a now obsolete version of Macintosh Common Lisp (MCL) and the PCL implementation of CLOS, which has been developed at Xerox Palo Alto Research Center. Recently, we upgraded our layout system to a new MCL version (release 2) offering an own implementation of CLOS. The upgrade of our meta-level techniques is still pending, since this MCL version currently supports no meta-object protocol.

7 Related Work

Events of interest are also relevant for visualization components of tracing systems. Boecker and Herczeg [3] present a system for defining tracing events. Instead of offering instance-specific meta-objects, a filter technique is used to simulate instance-specific behavior and

integrate general predicates into event definitions. See also Linden [25] for a discussion of visualization and filter definitions.

The Symbolics™ programming environment Genera™ offers also means for specifying layout of windows [32]. 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 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 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 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 were proposed which use TEX-like layout schemes for user interfaces. They also use constructs such as boxes and fillers for expressing window layout. The InterViews System [26] is a user-interface toolkit based on X windows and implemented in C++. Fillers and boxes are implemented as objects whereas our system uses virtual boxes and fillers. 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 a 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.

8 Summary and Future Work

This paper presented a framework for visualizing object-oriented systems. It consists of 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 notation are its expressiveness, user-predictable layouts, and efficient implementation schemes. The CLOS meta-level architecture is used to associate visualization and application objects. Supported techniques are indirect values, slot and method demons, and instance-specific meta-objects. These visualization techniques require no source code modifications to the systems selected for visualization.

Next steps might be to combine the advantages of TEX-style notations with the general flexibility of constraint systems. Another useful extension might be to support local variables in box specifications (see section 2.4). We also plan to address the problem of interpreting compound events (see Section 5.5) in more detail. Furthermore, research is necessary to extend this approach to 2-1/2 or 3-D layout.

Acknowledgements

We are grateful for helpful comments from the anonymous referees. The first author has been partly supported by a DAAD fellowship 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, M. Stefik, The LOOPS Manual, Xerox Corporation, December 1983.

3. H.-D. Böcker, J. Herczeg, What Tracers Are Made of, 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), 89–99.

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

5. M.H. Brown, Algorithm Animation, ACM Distinguished Dissertations Series, MIT Press, 1988.

6. Common Lisp Interface Manager, Version I, Release 0.9, Reference Manual, August 1990.

7. 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.

8. R. Cunis, Some Ideas on Integrating Reflective Aspects into CLOS-type Object Systems, Internal Report, University of Hamburg, Computer Science Department, 1990.

9. W. Cunningham, K. Beck, A Diagram for Object-Oriented Programs, *ACM Sigplan Notices* **21**, 11 (1986), 361–367.

10. 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.

11. J. Ferber, Computational Reflection in Class-Based Object-Oriented Languages, *ACM Sigplan Notices* **24**, 10 (1989), 317–335.

12. E.P. Glinert, S.L. Tanimoto, PICT: An Interactive Graphical Programming Environment, *IEEE Computer* **17**, 11 (Nov. 1984), 7–25.

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

14. N. Graube, Metaclass Compatibility, *ACM Sigplan Notices* **24**, 10 (1989), 305–315.

15. 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.

16. V. Haarslev, R. Möller, A Declarative Formalism for Specifying Graphical Layout, In: [33], pp. 54–59.

17. P.E. Haeberli, ConMan: A Visual Programming Language for Interactive Graphics, *ACM Computer Graphics* **22**, 4 (Aug. 1988), 103–111.

18. D.D. Hils, Visual Languages and Computing Survey: Data Flow Visual Programming Languages, *Journal of Visual Languages and Computing* **3**, 1 (Mar. 1992), 69–101.

19. K.M. Kahn, V.A. Saraswat, Complete Visualizations of Concurrent Programs and their Executions, In: [33], pp. 7–14. See also *Technical Report* SSL-90-38 [P90-00099], Xerox Palo Alto Research Center, 1990.

20. G. Kiczales, J. des Rivières, D.G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

21. G. Kiczales, D.G. Bobrow, Common Lisp Object System Specification, Part 3: Metaobject Protocol, *Technical Report* (draft), Xerox Palo Alto Research Center, October 1990.

22. K. Kimbrough, O. LaMotte, Common Lisp User Interface Environment, Preprint, Texas Instruments Inc., July 1989.

23. M.F. Kleyn, P.C. Gingrich, GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views, *ACM Sigplan Notices* **23**, 11 (1988), 191–205.

24. D.E. Knuth, TEX and Metafont – New Directions in Typesetting, Digital Press, 1979.

25. L.B. Linden, Parallel Program Visualization Using ParVis, In: *Performance Instrumentation and Visualization*, M. Simmons, R. Koskela (eds.), ACM Press, New York 1990, pp. 157–187.

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

27. P. Maes, Concepts and Experiments in Computational Reflection, *ACM Sigplan Notices* (Oct. 1987), 147–155.

28. P. Maes, D. Nardi (eds.), Meta-Level Architectures and Reflection, North-Holland, 1988.

29. 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.

30. B.A. Myers, D.A. Giuse, R.B. Dannenberg, B. Vander Zanden, D.S. Kosbie, E. Pervin, A. Mickish, P. Marchal, Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces, *IEEE Computer* **23**, 11 (Nov. 1990), 71–85.

31. R.E. Roberts, I.P. Goldstein, The FRL Manual, AI Memo 409 Edition, MIT Lab., 1977.

32. Handbooks of Symbolics Programming Environment, 7A, Programming the User Interface – Concepts, Symbolics Inc., 1988.

33. Proceedings, *1990 IEEE Workshop on Visual Languages,* Skokie, Illinois, Oct. 4-6, IEEE Computer Society Press, 1990.

34. P.H. Winston, B.K.P. Horn, LISP, 3rd edition, Addison-Wesley, 1989.