# A Refinement Approach to Visualization[*]

**Allen Goldberg**
**Rafael Furst**
**Cordell Green**

Kestrel Institute
3260 Hillview Ave
Palo Alto, CA 94304
{goldberg, rafe, green}@kestrel.edu

## Abstract

In this paper we present the design for a visualization system appropriate for instantiation in a Software Development Environment(SDE). The design lets the user of the SDE *declaratively specify* the *data* to be viewed and its *visual rendering*, and from that the visualization system generates the visualization. Motivating this work is the acute need for SDEs that can present dynamically-defined visualizations from user-defined, declarative descriptions of the data and its rendering.

Our approach is based on the observation that a data rendering is a representation of an abstract data type. This paper applies our previous and ongoing work on data refinement to visualization. We adopt that work to yield a practical, implementable architecture for a visualization system. We illustrate the design on a number of examples.

**Keywords:** Software visualization, refinement

## 1. A refinement approach to visualization

### 1.1. Objectives and Approach

We seek a SDE visualization system in which complex and varied visualizations are generated from declarative user descriptions. Current approaches are not sufficiently flexible – the visualizations they generate are pre-defined. In our approach, the user formulates a query to define the data to visualize and *composes* a graphical presentation from primitive visualization elements.

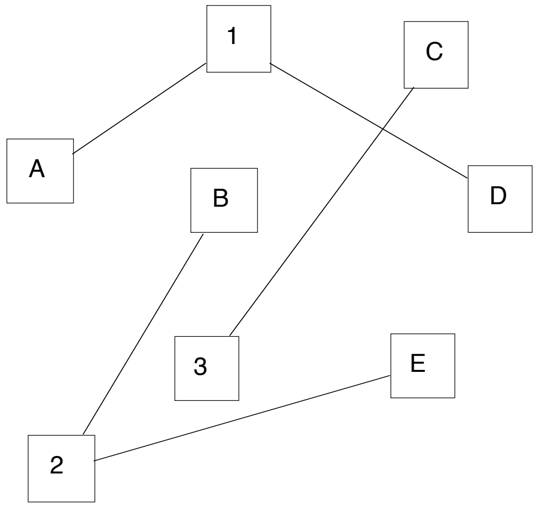
A simple expository example run through this paper. Assume a software development environment for modeling parallel and distributed systems. There is project database containing, among other things, a representation of the mapping of processes to processors, and, for a single program execution, the execution time of each process. Examples of the visualizations of this data that fall into the scope of our approach are illustrated in figure 1. As

[*] Work supported by Rome Laboratories on contracts F30602-94-C-0037 and F30602-93-C-0236.

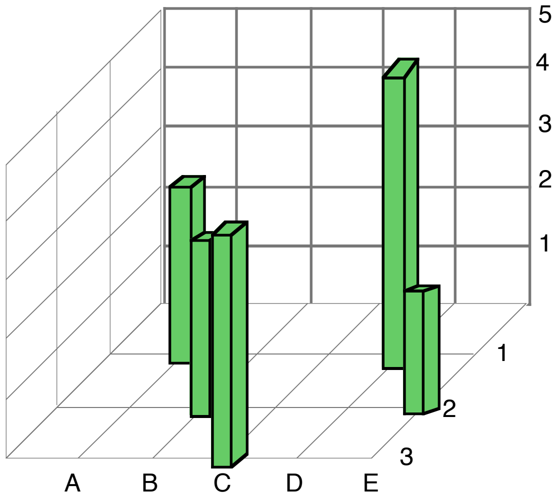
will become clear, these are just a few point solutions in an infinite space of generable renderings.

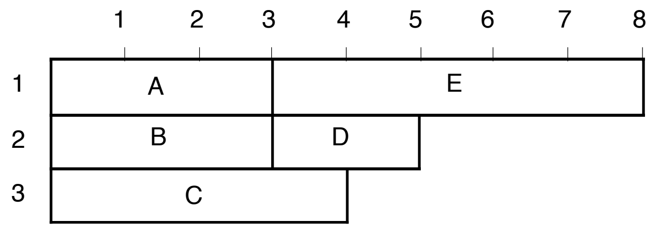
| Process | A | B | C | D | E |
|---|---|---|---|---|---|
| Processor | 1 | 2 | 3 | 1 | 2 |
| Time | 3 | 3 | 4 | 5 | 2 |

**1.A**



**1.B**





**1.C** **1.D**

**Figure 1.**

Our approach is based on the observation that a rendering ***is*** a representation of an abstract data type (with a restricted set of operations). Thus, a sufficiently general methodology for constructing data representations may construct renderings. We have done extensive work on the *refinement approach* [BlGo91, SrJu95] to data representation. This paper applies that work to visualizations. Our approach to data representation is *compositional:* data representations are constructed as compositions of modular refinements.

Kestrel's work on data refinement [BlGo91, SrJu95] as well as broad experience with specification techniques that use set theory [Jo86, Sp88], has shown that an effective refinement methodology is to refine domain-specific abstract types into domain-independent set-theoretic data types, and then refine the set-theoretic types to implementation-level representations. Because of the universality of set theory, the refinement of domain-specific types to set-theoretic ones is generally direct. Hence, the lion's share of the refinement task is cast as the problem of refining set-theoretic types. Since there are just a few standard set-theoretic types, an elaborated refinement or rendering methodology for these types is highly reusable across all problem domains.

## 1.2. Views: Extraction and Rendering

Most visualizations systems, and certainly those for SDEs, operate in an environment where there is a large data or knowledge base of "project data." Visualizations render some subset of data from the data base. Thus we separate the visualization process into two steps. First, *extract* the data being visualized as an instance of a *view*, where the term view is used in the same sense as it is in database terminology. Then *render* the view by associating graphical features to attributes of the view.

Formally, a *view* is a data type plus a value of that type. We call the data value the *extension* of the view. Based on the considerations above, the types used in views are set-theoretic types. The representation of domain-specific types as set-theoretic types is part of the extraction step. A view type is denoted by a *type expression* formed inductively from base types and type constructors. The base types are standard atomic data types: booleans, finite enumerations, integer and reals, and characters. The type constructors are tuples (records without named projection functions), sums (discriminated variant records), sets, sequences, relations and maps. The semantics and the operations associated with these types will be described below.

The visualization system must provide an interface for a user to query the project database. The query determines the view i.e. the data (the view extension) , and the type of the view. As a result of the extract step, the view extension is represented within the visualization system proper, independent of the project database.

Computation of the view extension depends upon the form of, and interface to the SDE project data base. It requires the formation and execution of queries to the SDE data base, and conversion of the data into the distinct representation used within the visualization system. SDE databases are often represented as persistent object base or a relational database. We note relational and object-oriented data have direct representation as set-theoretic types. While view extraction is non-trivial, due to its dependence on the specific underlying SDE database technology and representation schemes, we can only make few specific comments about view extraction. Thus the focus in this paper is on a methodology to render views.

View rendering depends on the view type but **not** the view extension. Our goal is to demonstrate that a small number of view types and associated renderings can be composed to generate a comprehensive set of visualizations. Key toward achieving this goal is a composition principle: **Rendering is accomplished by recursive decomposition of the type expression**. That is, to render the view with type expression `tc(t)` where `tc` is a type constructor and `t` a type, the renderings of `tc` and `t` are selected and composed. It is this composition principle that provides the flexibility for creating unique, on-the-fly visualizations from a fixed set of renderings.

## 1.3. Graphics Model

The visualization assumes a graphics subsystem providing an interface similar, but richer, than that of a typical drawing program such as *xfig*. It provides a viewing surfaces that displays either two dimensional or three dimensional objects. A coordinate system (polar or rectangular) is specified.

On the surface the usual types of graphical objects may be drawn. This includes various boxes, text, lines, polygons, rectangular solids, coordinate axis, etc. Execution of the rendering procedure generates a sequence of graphics commands to define a surface, and draw objects. The *properties* of the graphic objects, such a position, fill, and font size, must be determined by the rendering procedure.

## 1.4. Example

The visualization in Figure 1.A tabulates the assignment of processes, represented by a character, to processors represented by a integer and provides timing data for one program execution. To extract the data from the project database a query is formulated to generate a view of type `set(triple(char,int,int))`. The rendering in 1.A is the result of a composition of the following rendering functions for the type and type constructors forming the above expression: `set(_)`, `triple(_,_,_)`, `char`, `int`.

The rendering function for `set(_)` used in 1.A enumerates each element of the set and defines for each element a bounding rectangle of size 1 by 3 units on a 2 dimensional surface. Each bounding rectangle is positioned to the right the previous. For each set element it calls the rendering function selected for the element type (in this case `triple(_,_,_)`) to render the element. The rendering function for the elements is constrained to draw objects within the bounding rectangle corresponding to the element.

The rendering selected for `triple(_,_,_)` is similar. It divides the bounding rectangle into three bounding squares and calls rendering functions for each of its components. The rendering function for the components are constrained to draw objects within the bounding square. In this rendering the graphical property of *adjacency* captures the relationship between the three components of the tuple. The renderings of the `char` and the two `int` types are as 1 by 1 square whose text labels are the character and integers respectively.

It is often the case, that the data visualized is relational data. By viewing a relation as a `set` of `tuple`, we encapsulate renderings of relations as compositions of these constructors. An alternative rendering of a `pair` (`pair` is a specialization of `tuple`) is as two rectangles and a line connecting them. Another is to use the second element of the pair to define some property (such as size) of a graphical object representing the first. These alternatives and some others discussed in more detail below, neatly account for most renderings of relational data.

## 2. Detailed Description

## 2.1. Data types used in views

In this section we identify a collection of set-theoretic types and renderings of those types that is sufficient for capturing a wide-range of visual presentations. As is familiar from most programming languages, data types are generated from a collection of primitive types and type constructors.

Our type epistemology includes subtyping. For example, `map(x,y)` is a subtype of `set(pair(x,y))`. This means that any rendering of `set(pair(x,y))` is applicable to `map(x,y)` but other specialized renderings for `map` (i.e. ones that exploit the single-valuedness property that distinguish maps from arbitrary binary relations) are definable as well.

The data type operations are invoked to build the view extensions during the view extraction step and by rendering procedures to access the view extension. The operations required to support these activities is a subset of those used in a more general computational setting. For expository purposes, we classify operations into three categories: constructors, destructors and size operations. Generally, constructors are used “build” the view extension during extraction, and destructors are used to enumerate composite objects, such as sets, during rendering.

We assume each type has associated with it size function which maps element of the type to a positive number and a linear order which orders elements of the type.

The size function is useful because many visualizations represent numeric values as a numeric-valued property of an object, for example, the length of a rectangle. These operations are also be used to compute summary data, scale objects, and compute axis labels. The size functions listed in the table are defaults that may be explicitly overridden.

The linear order is used to sort values for presentations, so, for example, the elements of a set may be displayed sorted with respect to the linear order on the elements. All of the primitive types have a natural linear order. This is extended to all types inductively. For example sequences are ordered lexicographically in terms of the linear order on its elements. As with the size function alternative orders may be specified.

**Primitive Types**

The table below gives the primitive types that are used for visualization. Renderings for each type will be discussed in detail below.

| Type | super type | constructors | destructors | size | renderings |
|---|---|---|---|---|---|
| Integer | | integer constants | string rep | identity | property<br>string |
| integer subrange | integer | integer constants | string rep | identity | property<br>string<br>enumerations |
| real | | real constants | string rep | identity | property<br>string |
| enumerate-ion | | constants<br>val | ordinal<br>string rep | ordinal | iconic<br>finite property<br>integer<br>string |
| Boolean | enumeration | true, false | string rep | ordinal | iconic,<br>binary attribute |
| string | | string literal | | length | string |

The types described in this table are standard. We comment on the less obvious entries in this table. In a standard abstract datatype presentation of integers, the usual constructor functions are the constant zero and the successor function. In this setting a more appropriate constructor functions are all of the constant values of the type. This is because we are not interested in viewing integer or any of the primitive types as being composed from smaller or simpler values of the type. Similarly, the destructors simply map values to string constants that may display the value. Enumerations are finite types of user-specified values. The values of each enumeration type is in a one-one correspondence with integers via the functions val, and its inverse, ordinal.

**Type Constructors**

| Type Constructor | super type | constructors | destructors | size | renderings |
|---|---|---|---|---|---|
| tuple | | make-tuple | project | sum of the size of each component | juxtaposition of components<br>component values determine properties of a common object |
| pair | tuple | make-pair | first<br>second | | line-linked pair |
| triple | tuple | make-triple | first<br>second<br>third | | labeled linked pair |
| Set | | add-element | select-element<br>rest | sum of element size | region for each element |
| n-ary Relation | set | add-tuple | sel-tuple<br>rest | size | same as set(tuple) |
| Binary Relation | n-ary relation | add-pair | sel-pair<br>rest | size | graph |
| map | binary relation | ptwise extend | sel-dom-range-pair | size | same as<br>set(pair) |
| Seq | | prepend | first<br>rest | size | map(1..n, elem) |
| union | | mk-sum | | size of embedded value | union of renderings |

A tuple of size *n* is an aggregation, analogous to records, but without  component names. Pairs and triples are tuples with two, or three components respectively. Constructors construct the tuple form its components, and projection operations recover the components.

## 2.2. Rendering

A view identifies the data to be visualized as the view extension, and the form of the data as a type expression.  Rendering is the process of taking such a view and displaying it on the screen. In this section we describe how renderings of types and type constructors are defined, composed, instantiated and executed to create visualizations.

### 2.2.1. View Extraction

The view type, which is determined in the extraction step, determines the gross characteristics of the rendering. The same data may be represented by different view types because of "type equivalences." An example of such equivalences is functional currying: there is a one-to-one correspondence between values of type `map(pair(A,B),C)` and values of type `map(A,map(B,C))`. Another class of equivalences deals with inverses. There is a one-to-one correspondence between values of type `map(A,B)` and values of type `map(B,set(A))`. The type `rel(A,B)` (a relation whose first component is of type `A`, and whose second component is of type `B`) can be alternatively formulated as a `map` whose domain is `A` and whose range is `set(B)`. By extracting the same data into different view types different renderings of the data may be generated.

### 2.2.2. Rendering Procedure

Associated with each primitive type and type constructor is one or more renderings. A rendering is a template of a rendering function that gets instantiated to yield a procedure which takes an extension of a view as input and computes a sequence of calls to graphic primitives that render the data. The template parameters gets instantiated by the user. The template parameters include the names of rendering functions that correspond to sub-type expressions, and specifications of properties of graphic objects created when the rendering function executes. For example, a parameter of the rendering procedure for `set(_)` used in example 1.A is instantiated to a fixed set of graphical objects that label the visualization, name column headings and display legends. Other parameters select the size and spacing of the sub-surfaces that are used to render the set elements, and identify the rendering function for the element type.

Selecting and instantiating view templates is the primary interface between the user and the system. Practical application of this approach requires a user interface design that streamlines this interaction. A suitable interface would specify template values graphically, analogous to defining a master slide in a presentation program or a template word processing document. For example to instantiate the set rendering template the fixed objects can simply be drawn, and specifying the size and position of the bounding boxes of the first two elements would determine parameters that related to element positioning.

### 2.2.3. Rendering of primitive types

**Integers and reals**

The most elementary rendering of an integer or real is a text object in which the string representation of the integer value is rendered. Another alternative is to render it is an graphical object, say a rectangle, for which the integer value determines the length of the rectangle. Such a choice would be appropriate for rendering a single bar of a bar chart. More generally a rendering of an integer can be as an object in which the integer value determines any numeric property of the object – its distance along some axis, line thickness, font size, color hue or saturation, etc.

Unlike the renderings describe above, many renderings of an integer do not create a new graphic object but determine a property of an existing object which is passed to it as a parameter from a calling rendering function. For example to render a `set(pair(int,int))` as a scatter chart, a cross hatch object is associated with each pair in which each integer component of the pair determines the *x* and *y* position of the cross-hair object.

**Subranges, enumerations and Booleans**

The treatment of these types is similar to integers. They may be rendered as a print string. They may be rendered as an object in which the value determines some finite-valued property of the object such as fill, or font . Alternatively, the value may determine a finite-valued property of an existing object passed to it from a calling rendering function.

In addition, an enumerated type may be rendered iconically – a distinct  icon is associated with each value, with or without a legend.

Boolean is an instances of an enumerated type with two values.

### 2.2.4. Renderings of type constructors

**Tuples, pairs, and  triples**

A primary rendering of a tuple is to render each component of a tuple in a spatial arrangement, most typically using as adjacency, as in example 1.B, to express that each component is part of a single aggregate. A second rendering of a tuple is as an object in which each component value determines a property of an object, a rendering used in the scatter chart example. A pair may be rendered as two objects, linked by a connecting line. A triple map be rendered as a labeled link where two of the components are connected by a line and the third determines the thickness,  labeling, arrow head, or some other property of the line.

**Sets and Sequences**

A primary rendering of sets is illustrated in example 1.A. In that rendering a bounding box is defined and the rendering of each set element is constrained to fall within the box. There are many variations on this scheme which can be specified by instantiation of the rendering template. The bounding boxes may be arranged along different axis corresponding to a horizontal or vertical enumeration. The bounding box may be bounded along in one dimension but unbounded in others. The size of the bounding box may be scaled to the ratio of the size of the element to the sum of the sizes of all the elements. Another variation on this is to render the elements in an order that is sorted with respect to a linear order on the elements. The rendering function also provides for rendering of global objects that are not associated with individual elements. These global objects can be used to title the rendering.

While only one rendering of `set` is defined there are many renderings of the sets that the system can generate. First the set template has many parameters that lead to quite different visualizations (but keep the abstraction of rendering a set by rendering each element in a linear arrangement). Second a set may be rendered by its characteristic function and treated as a map. This is an example of a type equivalence.

The rendering of sequences follows the same pattern as those of sets, but the elements of the sequence are rendered in sequence order. A sequence of type `seq(t)` can be rendered as `map(int,t)`.

**Binary Relations**

The type `rel(A,B)` is a specialization of `set(pair(A,B))`. As such all of the renderings that can be composed from those types may be used to render `rel(A,B)`. In this section we present a rendering used in example 1.B that is specialized to binary relations.

Example 1.B, renders a binary relation as a graph in which the components of the relation are rendered as nodes. It cannot be rendered as a composition because values of types `A, B` are rendered as a unique nodes. Thus in the example even though the relation includes two tuples *(A, 1)* and *(D, 1)* there is only one instance of a node labeled *1*.  This factoring of common values into a single node is a property global to any single rendering function.

The rendering of a binary relation as a graph renders each value of component type `A`, respectively `B` via a recursive call to a rendering function for `A(B)` as an object and connects related objects with lines. The rendering procedure must incorporate a layout algorithm to constrain the position of object representing values of type `A` or `B`.

In the future we wish to expand the type system to include trees, directed acyclic graphs, and equivalence relations as specialized binary relations. Thus layout algorithms specific to these structures can be incorporated.

**Maps**

In this section we explain the basis of the visualizations examples 1.C and 1.D. These rendering present map data but do not use specialized map renderings. They do illustrate the robustness of the rendering presented so far.

To obtain the rendering of 1.C the data is extracted into a view of type `set(triple(char,int,int))`. Set is rendered on a three dimensional surface. Elements of the set are rendered in sorted order, using the default ordering on triples which is lexicographic order on its three components. The axis are laid out as the fixed part of the rendering for set. Determining the length and labels of each axis requires defining and computing size functions that differ from the default. For this technical reason specialized map renderings are defined to make convenient this common rendering.

Each element is rendered in a 2 dimensional bounded box whose *x* value is fixed to be one unit wide an whose *y* and *z* values are unbounded. Each set element is a triple which is rendered as a rectangular solid. Its width and depth are fixed to be .5 unit. Its length is the value of the third component of the triple Its *x* coordinates is determined by the position of the bounding box. Its *z* coordinate is zero. Its *y* coordinate is determined by the value of the second component.

To obtain the rendering of 1.D the data is extracted into a view of type `set(pair(int,(set(pair(char,int)))`. The rendering of set renders each pair within a semi-bounded box on a 2 dimensional surface. The width of the box is fixed along the y coordinate. The x coordinate is restricted to be positive. The elements of type `pair(int,(set(pair(char,int))` is rendered by constraining the objects of the pair to be rendered in adjacent bounding boxes. The first component of the pair, an integer, is rendered as a text object which displays the print representation of the integer value. The second component is of type `set(pair(char,int))`. This set is rendered as bounding boxes arranged along the *x* coordinate whose width in the *y* direction is fixed and whose length is determined by the value of the element. The elements of type `pair(char,int)` is rendered as a rectangle. The label of the rectangle is determined by the first component, the length of the rectangle, which is equal to the length of its bounding box, is determined as the value of the second component.

This example illustrates how the type of the view determines the gross characteristic of the rendering. It also illustrates the use of composition.

### 2.2.5. Composition Compatibility

If a rendering procedure is a composition of rendering functions (corresponding to subparts of a type expression) then some of the properties of a single graphic object may be determined by one rendering function, and others by another rendering function. This is a flexible model in which rendering procedures cooperate to define graphical objects and mutually “fill in” their properties. However, in this models not all compositions of rendering functions are valid. Specifically, a composition of rendering functions is *incompatible* if more than one function determines the same property of a graphical object, or if some property is not determined by any of the functions.

There are two interfaces between a rendering function *R* which calls a rendering function *S*.

- *R* provides *S* with a value to render, *S* renders the value as an object (or objects), but *R* constrains some of the properties of the object. Typically, *R* constrains the size, position or bounding box of the object. For the composition to be compatible *S* must not redefine the constrained properties. In example 1.A, each triple is constrained to be rendered within a separate fixed bounding box.

- *R* requires *S* to compute the value of a specified property of an object. For the composition to be compatible *S* must compute valid values for the property. In example 1.C, the rendering function for each triple requires the rendering function of its third component to compute the length of the bar representing the triple.

## 3. Related Work

### 3.1. Specware

Kestrel Institute is developing a system called Specware that supports a general paradigm of component composition and refinement. In this paper we have described work that has extended transformational approaches to automated visualization done at Kestrel [GrWe91] [WeGZ90]. We have used the Specware system in some simple experiments in which abstract types have been refined to graphical displays, partially validating the methodology described here.

There are differences between the design presented here and Specware. Specware relies on a powerful categorical operation known as colimit to horizontally compose abstract types. We are relying on a more restricted and conventional mechanism of polymorphic type constructors.

Using Specware refinements may be vertically composed: the rendering of an abstract type is definable as a sequential composition of refinements. In contrast each of our rendering functions completely describes a rendering of an abstract type as a single refinement step. The flexibility of sequential composition is that refinements can be factored to increase composability. For example one rendering of tuples uses juxtaposition to capture the connection among tuple components. The juxtaposition may occur in the x or y direction. The rendering can then be broken into a composition of two refinements: a refinement that expresses the design decision to use juxtaposition and another that selects the axis. Each of these individual refinements may be used in other contexts, increasing reusability. For example, the refinement selecting an axis may be used in a composition that defines a rendering of a set.

### 3.2. Other Work

Our approach is similar to that of Jock Mackinlay [Mack 86]. Mackinlay constructs graphical designs by matching data relations to primitives in a graphical language via a backtracking algorithm which considers various criteria such as effectiveness and expressiveness. We introduce the notion of a view which stands in a formal refinement relationship between the data and the graphical design. Rather than requiring that higher-level structures such as tables and graphs be part of the primitive graphical language, we attempt to synthesize these structures by exploiting the relational structure of the data in the recursive decomposition process described above.

A common approach to visualization in SDEs has been to hand-design general templates which can work with a large class of representations, such as source listings (e.g. [Eick 92]) or network structures (e.g. [KaKa 88]). In contrast, we attempt to derive visualizations on-the-fly which are specialized to, and take advantage of, structure in the data being displayed.

The work of Marc Brown [Brow 91] and others (e.g. [Roma 92], [StWh 93]) has concentrated on displaying dynamic aspects of computation (e.g. algorithm animation), but not the automatic generation of the visualization design. We anticipate extending our work and incorporating dynamic properties based on static representations derived in the manner presented in this paper.

Also related to our work is that of Chris Holt [Holt 94] on formalizing visual semantics.

## 4. Conclusions

The refinement paradigm is very powerful and its application to visualization is natural. Our experience with refinement in other programming domains, especially data refinement , suggests that the hard work is to formalize the design space of refinements (renderings) to maximize composability. This takes a combination of experimental work and analysis to identify the most useful abstractions.

## Bibliography

[BlGo 91] Blaine, L. and Goldberg, A. DTRE -- A Semi-Automatic Transformation System. Appeared in, *Constructing Programs from Specifications*. B. Möller, Ed., North Holland, 1991 Kestrel Institute Technical Report KES.U.91.3, May 1991.

[Brow 91] Brown, M.H., "Zeus: A System for Algorithm Animation and Multi-View Editing," *1991 IEEE Workshop on Visual Languages* (October 1991), pp 4-9.

[Eick 92] Eick, S.G., Steffen, J.L., and Summer, E.E., "SeeSoft Tool for visualizing line oriented software," *IEEE Transactions on Software Engineering,* 11 (18), 1992, pp. 957-968.

[GrWe 91] Green, C.C. and Westfold, S.J., "Synthesis of Tabular and Graphical Displays", Final Report of NOSC SBIR Phase II, July 1991.

[Holt 94] Holt, C.M., "An Algebra of Lines and Boxes," *Proceedings of the 1994 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, 1994, pp. 55-62.

[Jone 86] Jones, Cliff B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[KaKa 88] Kamada, T. and Kawai, S., "Automatic Display of Network Structures for Human Understanding," Technical Report 88-007, University of Tokyo, 1988.

[Mack 86] Mackinlay, J., "Automating the Design of Graphical Presentations of Relations Information," *ACM Transactions on Graphics*, Vol. 5, No. 2., April 1986, pp. 110-141.

[Roma 92] Roman, G-C., Cox, K.C., et al, "Pavane: a System for Declarative Visualization of Concurrent Computations," *Journal of Visual Languages and Computing* (1992) 3, pp. 161-193.

[Spiv 88] Spivey, J.M., *Understanding Z*, Cambridge University Press, Cambridge, 1988.

[SrJu 95] Y. V. Srinivas and Richard Jüllig, "Specware:™ Formal Support for Composing Software," *Proceedings of the Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, July 1995.

[StWh 93] Stasko, J.T. and Wehrli, J.F., "Three-Dimensional Computation Visualization," *Proceedings of the 1993 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, 1993, pp. 100-107.

[WeGZ 90] Westfold, S.J., Green, C.C., and Zimmerman, D.Z., "Automated Design of Displays for Technical Data", Kestrel Technical Report, June 1990.