Working Paper Series
ISSN 1170-487X

A graphical notation for the design of information visualisations

by Matthew C. Humphrey

Working Paper 97/5
February 1997

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

A graphical notation for the design of information visualisations

MATTHEW C. HUMPHREY

Department of Computer Science, University of Waikato, Hamilton, New Zealand
email: matth@cs.waikato.ac.nz , http://www.cs.waikato.ac.nz/~matth

Visualisations are coherent, graphical expressions of complex information that enhance people's ability to communicate and reason about that information. Yet despite the importance of visualisations in helping people to understand and solve a wide variety of problems, there is a dearth of formal tools and methods for discussing, describing and designing them. Although simple visualisations, such as bar charts and scatterplots, are easily produced by modern interactive software, novel visualisations of multivariate, multirelational data must be expressed in a programming language. The Relational Visualisation Notation is a new, graphical language for designing such highly expressive visualisations that does not use programming constructs. Instead, the notation is based on relational algebra, which is widely used in database query languages, and it is supported by a suite of direct manipulation tools. This article presents the notation and examines the designs of some interesting visualisations.

1. Introduction

Visualisations are graphical representations of information that enhance people's ability to communicate, analyse and reason. Detailed maps, weather histories and diagrammatic transportation schedules communicate prodigious amounts of information clearly and concisely [Tufte 83] [Tufte 91]. Scatterplots, scientific false-colour renderings and web diagrams encourage the discovery of structure hidden within information by making relationships apparent [Becker 87a] [Becker 87b] [Chernoff 73] [Cabral 89]. Pictorial examples facilitate learning, while graphical representations of design promote problem-solving [Carroll 80] [Shneiderman 83].

Although there are myriad tools for producing visualisations, they do not fully support the design of visualisations; they are caught in the struggle between power and ease of use. Current tools either limit graphical expressiveness or require the use of programming skills [Myers 87] [Kazman 96]. Powerful tools use programs or complex rules to specify the structure of new displays. They are not usable by people who lack programming skills. Tools that are easy to use produce only a few types of visualisation and do not allow the user to explore and create new display types.

To address these limitations, our research has developed and evaluated the Relational Visualisation Notation—a graphical language based on relational algebra for designing a wide range of complex information visualisations without programming. We hypothesise that the notation supports the visualisation design process effectively by encouraging the creation, development, assessment and application of novel visualisations.

This article presents the notation and examines the designs of some interesting visualisations. Section 2 outlines what we mean by information visualisation and presents several important limitations of existing visualisation tools. It also introduces the new notation and establishes its foundation in relational algebra. Section 3 presents *graphic relations*—visual counterparts to traditional database relations that act as visualisation building blocks. The notation itself is described in detail in Section 4 along with an example that recreates Minard's graphic. Section 5 describes the Relational Visualisation Toolkit and shows how someone

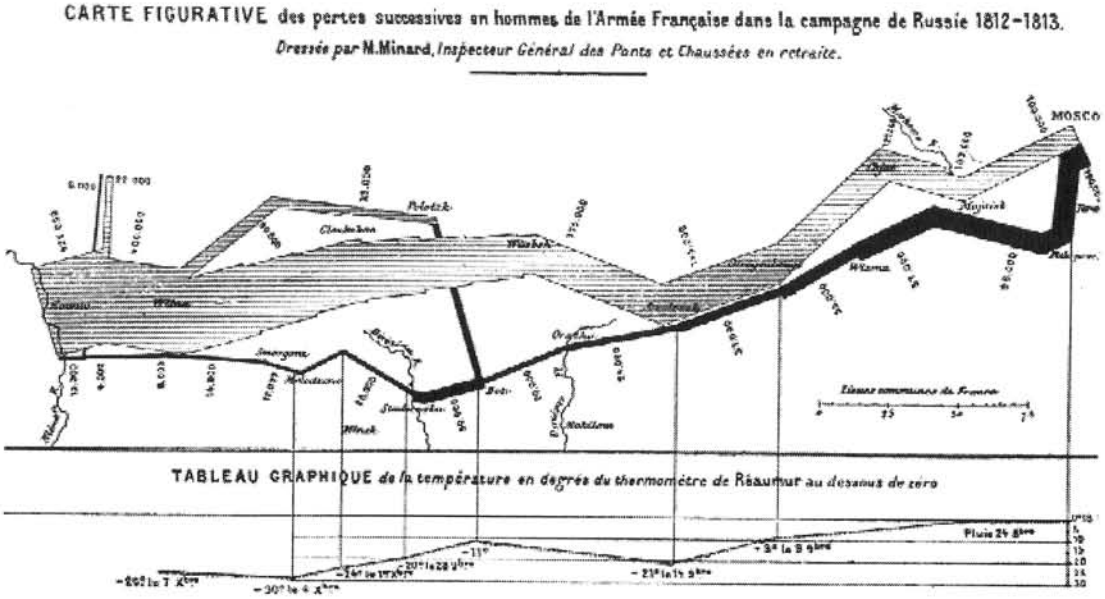

Figure 1 Napoleon's march on Moscow

might use it to create a novel visualisation. The benefits and limitations of using relational algebra to design visualisations appear in the conclusion in Section 6.

2. INFORMATION VISUALISATION

The essence of information visualisation comprises the data to be presented (in a semantic data model), the pictures that convey the data and the relationships between the data and the pictures [Bertin 83] [Kosslyn 85]. The best way to describe a visualisation is to show one. Minard's famous graphical account of Napoleon's 1812 assault on Moscow appears in Figure 1. The 422,000-strong French Army battles its way eastward across Russia, assails Moscow, and then retreats through the harsh Russian winter. Although Minard drew this by hand, computer-generated versions have also been produced [Roth 94] [Shaw 94] [Kazman 96].

In this figure there are formal data, such as the number of troops that arrived in a particular city, the temperature on a particular day and the location of the city on the map. There are graphic components, such as thick line segments, written text and numbers, vertical lines, and the map layout. And there are relationships, such as line thickness indicating troop size, shading indicating advance or retreat, names representing cities, and vertical distance encoding the temperature. Furthermore, nonformal information describing the movement of troops is indicated by a connection between cities, even though the time and direction of the movement are implied only by the line's continuity and Napoleon's devastating losses. Because this example aptly illuminates visual communication, we will return to it throughout the article.

2.1. EXISTING VISUALISATION DESIGN TOOLS

Current tools for designing visualisations exhibit several interrelated problems. Many lack flexibility; they do not provide enough alternatives for successful design exploration. Those tools that are the most flexible achieve their range of options through programming notations, which limit their usage to programmers. A few tools fall between these two extremes and provide some flexibility without programming. However, they often embody the worst of both worlds: limited flexibility coupled with a new, complex notation.

Some tools, such as those in spreadsheet packages, enable the user to easily explore a small number of visualisation styles, such as bar charts, pie charts and scatterplots. With a few key

presses, the user chooses the desired design. However, these tools provide no means for combining visualisations, or relating one part of a visualisation to another. They also use limited data models that cannot store the complex information found in realistic situations. The user is required to stay within narrow design specifications.

Most display design tools achieve flexibility through a general-purpose programming language [Rhyne 87] [Myers 89]. The visualisation design is a program. But programs make poor design specifications because they emphasise algorithmic structure rather than the structure inherent in the problem [Sommerville 93]. Consequently, they make the visualisation difficult to produce, difficult to modify, and difficult to debug. Visualisation programs are usually neither portable, reusable nor device independent.

Furthermore, these tools are usable only by programmers, or people with programming skills. Programming is the expression of an algorithm incorporating sequencing, decision and iteration. To program, one needs to be familiar with procedural abstraction, data structures, variables and memory state. In some cases recursion or recursive thinking is required. Creating visualisations by programming requires knowledge of graphics libraries and their protocols, rendering architectures, and other programmatic details. People with a deep understanding of their data, but little programming experience, cannot easily use these tools. Programmers, moreover, are generally not skilled in graphic display design.

Between the explorable, limited tools and the powerful programming tools are those that attempt to balance usability and flexibility. Their use of a constructional notation makes them flexible; visualisation designs are specified as diagrams linking data and transformation. They permit abstraction and allow components to be combined into more complex designs. They achieve a measure of usability by providing graphical notations supported by direct manipulation user interfaces. Designs may be expressed as annotated graphemes [Roth 94], flow diagrams [Upson 89], wiring diagrams [Ingalls 88], switchboards [Waite 91], or other visual metaphors.

However, these compromise tools also have inadequacies. They are still not powerful enough to specify novel, realistic visualisations. They limit the user to particular design categories, such as iconic or widget-based displays. Their notations are based on concepts that users are not likely to be familiar with, such as constraint propagation. New notations, especially when they introduce unfamiliar concepts, are difficult to learn and often resisted by users [Tse 91]. Programming architectures, such as object/message-passing, further limit the usability of some tools.

Sage is a notable exception to the current range of visualisation tools, as it both permits a wide range of visualisations to be described and does not make use of traditional programming structures. Sage supports the design process by allowing users to search a database of designs similar to the one they wish to create. Searching is automatic and based on similarities between the visualisation they are specifying and those in the database. Although Sage does not explicitly contain relational operators or querying mechanisms, it is capable of visualising multiple relations simultaneously. It can produce visualisations for N-ary relations that are specified either as a single relation, or as several binary relations.

Sage describes a visualisation design with primitive graphic objects called graphemes: label, mark, line, etc. These have attributes such as position, colour, thickness and so on. The user associates data attributes directly with graphic attributes. However, there do not seem to be any mechanisms for more complex expressions, such as logarithmic or trigonometric relationships, or dependencies on other graphic objects.

The system is sufficiently expressive to design a rendition of Minard's graphic. This version, shown in Figure 2, places the cities on a grid indexed by longitude and latitude. The troop path is shown as coloured segments for which the colour represents the temperature and the line width is the population. Diamonds mark the sites of important battles. The

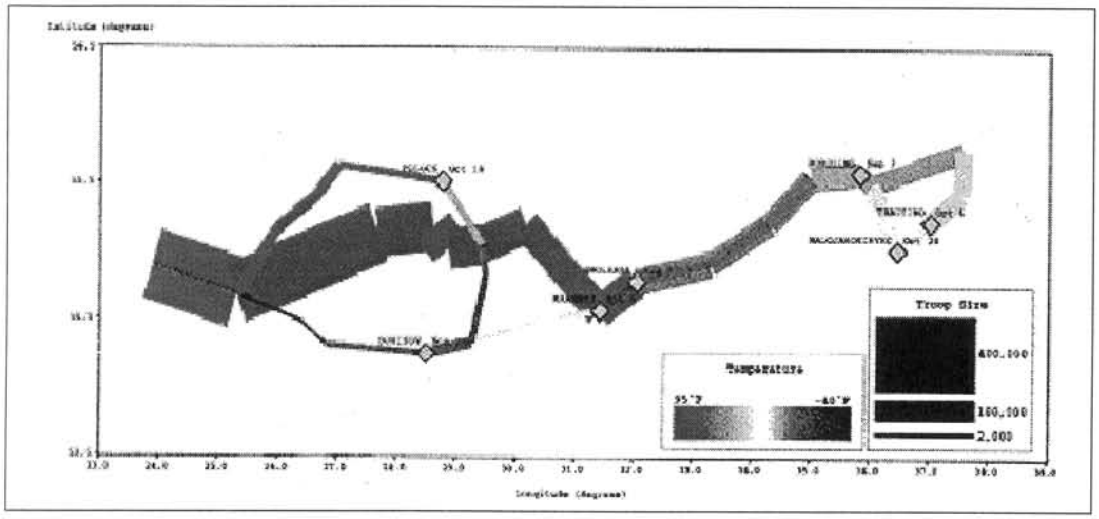

Figure 2 Minard's graphic via Sage

visualisation is created by associating information attributes directly with graphic attributes. Implied transformations, such as longitude to x-axis position and colour index to specific value, are constructed automatically by the supporting tool.

Sage does not use a formal definition of relations and has no user-accessible relational operators such as select or join. Graphemes associate with tuples sequentially—those appearing early in a relation generate marks before others. Diagrams with overlapping elements, such as Minard's graphic, are difficult because the order or drawing cannot be controlled except by sorting the data. Likewise, new information cannot be derived. Averages, sums, partial sums cannot be computed, except automatically as components of visualisations. For example, Sage cannot produce pie charts unless the wedge percentages have already been computed.

2.2. THE RELATIONAL VISUALISATION NOTATION

The Relational Visualisation Notation described in this paper allows users to specify visualisation designs graphically. The notation is composed of three main parts: semantic data models, graphic relations and design diagrams. The semantic data modelling component provides facilities for describing, storing and retrieving information according to the relational model—like a relational database. Each graphic relation visually represents one information relation. It defines the informational and graphical models of the visualisation as well as the transformation between them. Design diagrams combine multiple information and graphic relations into a visualisation design specification. All the components are supported with an integrated set of direct-manipulation design tools.

Graphic relations are defined by an information schema, a graphic schema and a set of arithmetic expressions. The information schema characterises the information to be presented by the graphic relation. The graphic schema visually represents the visualisation; it is made of boxes, circles, lines and other graphic figures including novel elements, such as graphic replication and selection groups. Users draw the graphic schema using a specialised graphic editor. The arithmetic expressions encode the relationships between the information schema and the graphic schema.

A visualisation design combines graphic relation designs with relational operators that manipulate the data of several relations. Design diagrams are directed, acyclic graphs that combine several source relations to produce several output graphic relations. The nodes

represent source data, relational operators (such as project, select and join), and reusable, hierarchically nested designs. Using direct-manipulation tools, users draw design diagrams that link data to relational operations and graphic relations. The design may then be used to produce a visualisation from data.

The intended users are people who work with data, but who also need graphical representations of that data. This includes scientists, statisticians, information specialists, database designers, knowledge engineers, and even user interface designers. Users have a thorough understanding of the data to be visualised, and accompanying skills in data modelling and manipulation. They should be able to create spreadsheet-like expressions, formulate relational queries and construct relational views. Naturally, they need to be familiar with geometric relationships that express information. However, they would not necessarily have any programming skills and should not have to learn any.

2.3. RELATIONAL INFORMATION MODEL

The information domain of graphic relations relies on a formal definition of data relations, as follows. Given a sequence of domains, $D_1, D_2, \ldots, D_k$, a relation **R** is a subset of the cross product of the domains. Formally, $\mathbf{R} = \{t \mid t \in D_1 \times D_2 \times \ldots \times D_k\}$. In other words, a relation is set of tuples, each being a sequence $<a_1, a_2, \ldots, a_k>$ of atomic data items. Every data item a_i belongs to domain D_i. All tuples in a relation have the same "arity" or number of attributes, and all tuple values in the same position belong to the same domain. A relation is characterised by a schema **S,** which is a sequence of ordered pairs $(s, D)_i$, s being the name of the attribute and D its domain: $\mathbf{S} = (s_1, D_1), (s_2, D_2), \ldots, (s_k, D_k)$. Relations are usually shown as tables in which the rows are tuples and the columns are attributes [Modell 92].

Attribute domains describe the range of possible values that the tuple values may assume. They resemble domains in a database or primitive data types in Pascal or other programming languages. Table 1 shows domains similar to those found in commercial tools, such as spreadsheet and database packages. Strings are sequences of zero or more printable and non-printable characters. Floats are approximations of real numbers. Booleans are either true or false. Bitmaps are rectangular areas of pixels. Many modern databases also include domains for dates, times, etc., which we do not include because they do not increase the power of the notation and can easily be added later.

A value must conform to its domain. For example, every value of a Boolean attribute must be either true or false. Tuple entries may not be null, although such values could be modelled with a "missing attribute" Boolean attribute. A true value indicates that the attribute exists, while a false one implies that the attribute is missing, even though it contains a value appropriate to the domain. Relations themselves may not be used as domains, even though finding ways of doing so is an important area of theoretical database research [Tsichritzis 88].

The key of a relation, which is the attribute or set of attributes that uniquely identifies each tuple, is taken to be the entire attribute set and no other assumptions are made about the existence of primary keys. Normalisation, consistency and integrity maintenance are irrelevant because the graphic relation algorithms only read the database and do not update it. The visualisation techniques do not modify the information of the source data relation.

Domain name	**Range of values**
Integer	Positive or negative integer values
String	Alphanumeric strings of varying length
Float	Floating point real number
Boolean	True or false value
Bitmap	Varying-sized bitmap picture

Table 1 Data domains

Tuples in a relation are unordered and it is not possible to rely on them occurring in any particular order. They may even appear in a different order from one access of the database to another. In any case, graphic relations do not process tuples sequentially; instead they define an order-independent mapping from tuple attributes to graphic attributes. Although many commercial databases model relations with lists, no consensus exists on how to preserve ordering within queries or from one query to the next. To avoid these problems and to maintain mathematical integrity, we assume relations are unordered.

3. Graphic relations

Visualisations can be considered to consist of several distinct *graphic relations* layered together. A graphic relation represents relational information visually. It communicates the information content of a relation with pictorial instead of textual language—each data tuple corresponds to a *graphic tuple*. A graphic tuple is complex symbolic mark whose geometric, retinal and iconic attributes encode attributes of the data. The image produced by the collection of marks is equivalent to the source data relation.

A graphic relation is generated from a graphic relation *design*, which specifies a class of similar images. When applied to a particular instance of a data relation, it produces a graphic relation image. A graphic relation design consists of an information model, a graphic schema and arithmetic expressions. The information model describes the schema of the relations that can be represented.

The graphic schema consists of graphic objects, such as rectangles, circles and bitmaps, hierarchically composed with graphical grouping objects. Graphic groups, such as replications and selections, increase the expressive power of graphic relations by allowing graphic objects to be collected, replicated and selected dynamically—as the image is being generated. Attributes of the graphic objects, such as position and colour, are computed by arithmetic expressions that use information from the input tuple and attributes of other graphic objects. The syntax of these expressions is similar to spreadsheet formulas.

3.1. AN EXAMPLE OF A GRAPHIC RELATION

A data relation for describing bolts is used throughout the upcoming sections to describe the information structure of graphic relations. Table 2 gives the schema and data of the relation—a typical parts inventory of metal fasteners. The tuples describing the bolts are distinguished by a primary key *part Id*. Bolts come in a variety of lengths, measured in millimetres, as indicated by the attribute *length*. Attribute *finish* contains one of the two types of finish: brass or zinc plated, denoted by a text string. The attribute *tpi* stands for the number of turns per inch, which is used for bolt-nut compatibility. The *cost* is measured in cents per bolt.

Figure 3 is a graphic relation showing information about bolts. The eight data tuples are

part Id Integer	**length** Integer	**finish** String	**tpi** Integer	**cost** Float
100	30	brass	8	0.04
101	20	zinc	8	0.04
102	20	brass	16	0.08
103	20	zinc	16	0.08
200	30	brass	8	0.10
201	30	zinc	16	0.12
202	40	zinc	16	0.12
203	40	zinc	12	0.16

Table 2 The Bolts relation—schema and data

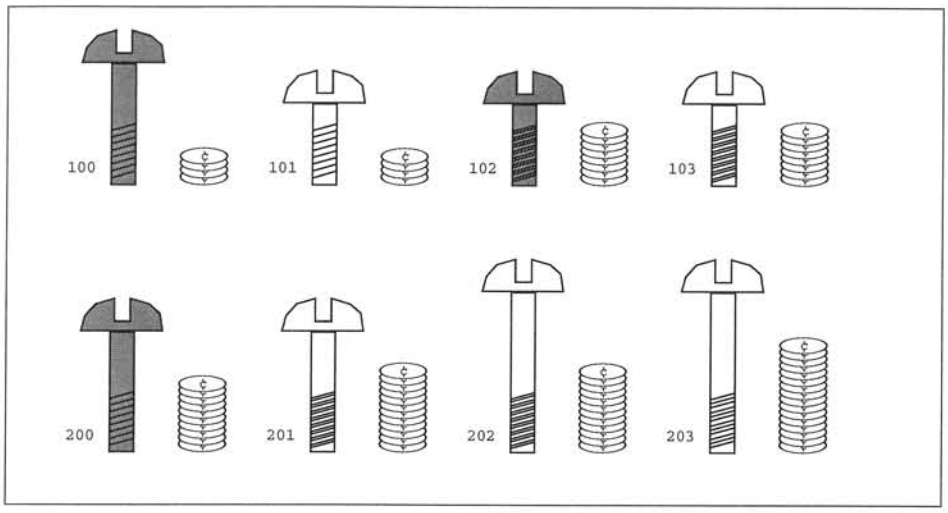

Figure 3 Visualisation of the bolt relation

transformed into eight graphic tuples that each comprise a parameterised icon: a bolt with a stack of coins and a part number. The height of a bolt in the picture is proportional to its length in the data, and the number of coins in the stack is its cost in cents. The horizontal position of each bolt is derived from its part number. The diagonal lines on the bolt represent the number of turns per inch, and are proportionally spaced so that higher turn counts are increasingly dense. The exact number of lines is the number of turns per inch. The shading of the bolt indicates its finish, grey for brass and white for zinc.

The design of the bolts graphic relation is shown in Figure 4. The information model is the schema of *Bolts* and specifies the set of possible data items that can be represented. The graphic schema is a template of rectangles, ellipses and text that is instantiated for each tuple. The dotted rectangles and names identify discrete objects of the graphic schema. The arithmetic expressions relate *part Id* to the x- and y-positions of the image, *length* to the vertical height of the rectangle, and so on for other attributes.

As Figure 4 illustrates, the line drawing of the bolt will represent each bolt in the database.

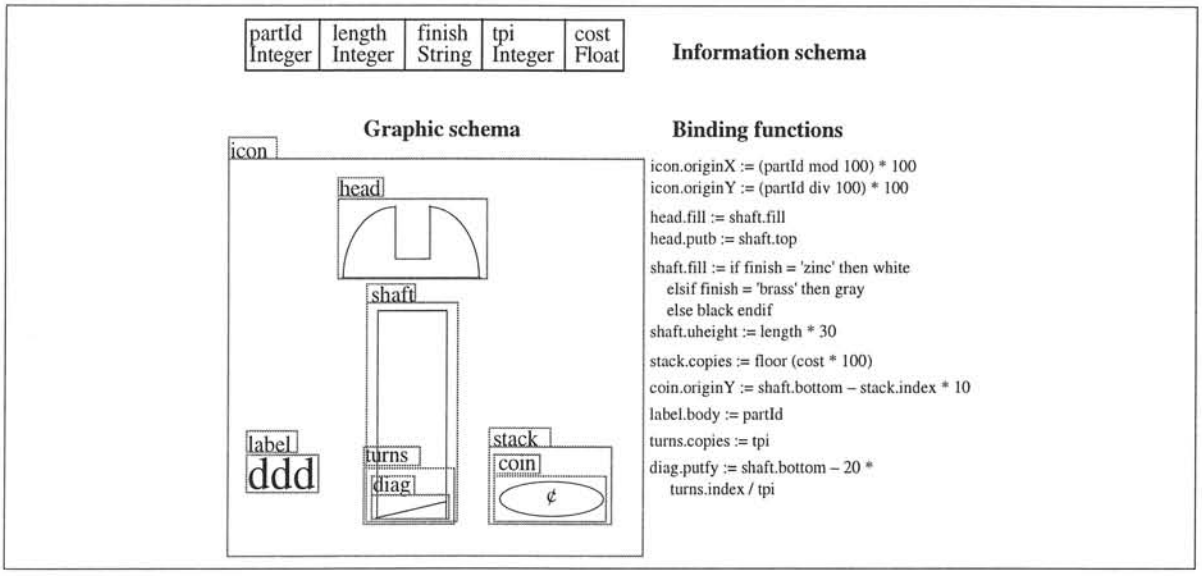

Figure 4 Graphic relation design

Figure type	Mark shape
Rectangle	Filled or outlined rectangle or square
Ellipse	Filled or outlined ellipse or circle
Wedge	Filled or outlined arc from an ellipse or circle
Line	Straight edge between two points
Polygon	Filled or outlined convex, concave or complex polygon
Text	Fixed-width block of text in a single font
Bitmap	Varying-sized array of pixels

Table 3 Primitive graphic objects

When instantiated, its length will vary with the *length* specified in the relation, and its shading will become the *finish*. The text string "ddd" shows where the *part Id* will be placed. The ellipse represents a coin, copies of which will be stacked up according to the *cost*, in cents. The diagonal line will be replicated and spaced proportionately for the screw threads.

The graphic relation is produced by an instantiation process that generates one graphic tuple for each data tuple in the relation. Information in the source data relation instantiates the graphic schema, which is replicated for each tuple. The arithmetic expressions customise the template, which is then displayed. When instantiated with the tuple, the graphic schema becomes a set of graphic objects. The attributes of the schema's objects are computed from the expressions. Repeating this process for each tuple constructs the graphic relation. The resulting picture appears as the objects are drawn. Each tuple in Figure 3 is produced one at a time by instantiating the graphic schema with the values from the information model.

3.2. GRAPHIC SCHEMA

A graphic schema is a collection of primitive graphic objects, summarised in Table 3, and graphic grouping objects, described in Section 3.4. Each type of object corresponds to one type of figure that can be drawn on whatever visual medium is used. Although the effectiveness of the graphic relation technique does not depend on the specific choice of primitives, a wide range of graphic objects is preferred. We have chosen objects similar to those found in many popular graphics systems.

Objects have geometric, retinal and iconic canonical attributes. Geometric attributes are further divided into spatial and positional attributes. Spatial attributes modify the coordinate system in which the object is to be drawn, relative to rotations, translations and scalings of the current coordinate system. Positional attributes are those that describe the object's geometry within its own coordinate system. Object features, such as fill pattern, texture, line width and text contents, are described by retinal attributes. Iconic attributes indicate the alternative shapes that marks use to represent information. Iconic structure and variability is discussed with the graphic grouping objects in Section 3.4.

Table 4 shows the attributes of each type of object. Similarly named attributes have similar functions across different objects, and blank entries indicate that the object does not have that particular attribute. The final column indicates the domain of each graphic attribute. As in other graphics systems, spatial attributes are common to all objects [Adobe 85] [Mallgren 83] [Hopgood 92].

The *originX* and *originY* specify the distance by which the origin (0,0) of the object's coordinate system should be translated. The new, translated space may be rotated about its origin; rotation values are specified in radians. After rotation, the coordinate space may scaled independently in the x and y directions. Scale values greater than 1 cause the objects to appear larger and values less than 1 make them appear smaller.

	Rectangle	Ellipse	Line	Text	Polygon	Wedge	Bitmap	Domain
Spatial Attributes	originX originY scaleX scaleY rotate	originX originY scaleX scaleY rotate	originX originY scaleX scaleY rotate	originX originY scaleX scaleY rotate	originX originY scaleX scaleY rotate	originX originY scaleX scaleY rotate	originX originY scaleX scaleY rotate	Float Float Float Float Float
Positional Attributes	left top right bottom	left top right bottom	xfrom yfrom xto yto	left top right	left top	left top right bottom	left top	Float Float Float Float
Iconic Attributes				text	numPoints closed points	startAngle endAngle	contents	String Integer Boolean String Float Float Bitmap
Retinal Attributes	filled fill edge thickness	filled fill edge thickness	edge thickness	filled fill edge thickness font pointSize justification	filled fill edge thickness	filled fill edge thickness	String	Boolean Bitmap Bitmap Integer String Integer

Table 4 Graphic object attributes

The values *left*, *right*, *top* and *bottom* define a bounding rectangle that tightly encloses its rectangle, ellipse, text or wedge. Text is placed in a rectangle whose depth is determined from the amount the text. The bottom attribute, therefore, is not canonical for text; the top position of the text determines where the bottom will be. Line segments connect the point at *xfrom*, *yfrom* with the point at *xto*, *yto*. Polygons and bitmaps have a single point of reference that positions them as a whole unit.

Text, polygons, wedges and bitmaps have several iconic attributes that define their shape. The body of a block of text is a string parameter. The number of data points in a polygon is accessible as an integer along with a text string *points* containing x-y pairs such as "x1,y1; x2,y2..." and so forth. (Polygon points are defined in terms of a formatted string, rather than a data structure, because the data must be available to the arithmetic expressions, which can only manipulate simple domains.) The *closed* value indicates whether the figure should be a closed loop or an open jointed line segment: only the former can be filled. The wedge is defined in the clockwise direction from a starting angle to an ending angle. The angles are measured with 0 radians being out to the right (3 o'clock) and proceeding in the positive direction clockwise (6 o'clock is $\pi/2$) and in the negative direction anti-clockwise (12 o'clock is $-\pi/2$). The complete contents of a bitmap can be assigned by parameter.

Many objects may be filled with bitmap texture patterns by specifying the *filled* Boolean attribute. The corresponding *fill* attribute specifies the pattern with which the object should be filled. Several texture patterns are defined through constants, including *Grey, DarkGrey, Black, White, LightGray* and others. The *edge* attribute also specifies a texture pattern with which the borders of an object should be drawn. Lines themselves cannot be filled; they have only edging texture. The *thickness* attribute determines the width of the outline edging of the figure. A zero thickness indicates that there is no edge.

Text has additional attributes. The *font* attribute specifies a named font, such as "Courier", "Times", "Helvetica", etc. Likewise, *point size* is a positive integer that sets the size of the font. *Justification* is a string determines how the text is to be formatted within the rectangle. The value "left" is for left justification in which all text is flushed to the left edge, "right" is for

right justification, "centre" for centre justification and "both" for fully left and right justified text.

We have chosen this particular set of attributes because it meshes easily with the graphics system we are using. There is nothing special or unique about these attributes; some are rarely used, and others that have been omitted may be needed frequently in other circumstances. The use of different visual media would also require slightly different sets of attributes. For example, if three-dimensional graphics were available, then attributes for the third dimension would need to be added. Colour is not present here because the system has been implemented using monochrome graphics. It would be straightforward to add colour.

3.3. ARITHMETIC EXPRESSIONS

An arithmetic expressions associated with a graphic object provides a means for varying an attribute of that object based on information in the data tuple and in other graphic objects. We say that the attribute is *bound* to data values that occur in the expression. The expression binds the input values to the destination attribute and takes the form of an assignment statement: *object . attribute-name := expression.* The left hand side describes the destination of the result; it contains the name of the graphic object and the name of the attribute, separated by a dot. The right hand side contains the functional expression. Although examples show binding expressions, or just *bindings*, as assignment statements, the expression language contains no assignment operator. In the Relational Visualisation Toolkit, the user need specify only the expression body.

The expression language is similar to traditional arithmetic or spreadsheet formulas. An expression may contain literal values, named constants, attributes of the input data tuple, references to attributes of other objects, calls to pre-defined functions as well as ordinary arithmetic operators. The language also includes a value-returning if-then-else function that selects a result expression based on a Boolean value. It may be cascaded to produce arbitrary data selections.

Furthermore, expressions only compute new values to be assigned to attributes; they cannot produce side effects. The only effect that the evaluation of an expression can produce is the modification of the graphic objects. Figure 4 shows several graphic attributes bound to arithmetic expressions. Evaluating the expressions in the context of each data tuple instantiates a new graphic tuple. The resulting modified graphic objects are shown in Figure 3.

3.4. GRAPHIC GROUPING OBJECTS

Graphic groups provide hierarchical structure for the graphical schema. They compose primitive objects into single units, which can then be used in other groups. They also provide a means for statically specifying variability in the graphical structure. Together with binding expressions, they can produce graphical images with different numbers and kinds of components.

There are three kinds of graphic group: collections, replications and selections. A graphic collection organises its components into a single unit—its elements can be rotated, translated and scaled together. Like a "group" within popular drawing packages, it can be dragged, sized, and manipulated as a single object. A graphic replication duplicates its components a fixed number of times, as specified by an integer. A graphic selection only draws one of its components, as chosen by an integer.

The graphic groups can create hierarchies. Any group or graphic object may be used as an element of another group. For example, a collection can have several primitive objects, plus two replications and a selection. The replication may contain a selection. The former selection may itself contain other primitive objects and another replication. No graphic object of any

kind can be an element of more than one group. Ultimately, every object is a member of exactly one group, except for the root collection, which stands alone.

The attributes of these objects are shown in Table 5. All graphic grouping objects have spatial attributes that modify the coordinate system of their components. Each group can translate, scale and rotate the coordinate system that it inherits from its parent. It then passes the new system to its children. Collections have no other attributes. Replication groups have a duplication count that indicates the number of copies they are to produce and an index that identifies each unique copy. Selection objects have a choice value that determines the component to be drawn.

A group's subobjects are fixed for a particular design—they can only be modified while the graphic relation is being created and edited. No attributes refer to the components, either for reading or updating. Binding expressions cannot index components individually, delete them, modify them, or add new ones.

The controlling attributes of graphic replications and selections are intended to be bound to expressions. A replication group produces a varying number of images for every input tuple when a binding expression specifies the number of copies. A selection group draws different graphic objects for tuples with different values when a binding expression specifies the selected component. By appropriate bindings, the structural shape of a mark can vary with the input data.

We digress here and mention the order of drawing because it is relevant to groups. Within every group, including the root collection, graphic objects are drawn in a particular sequence in order to build up a two-dimensional image. Graphic objects closest to the background are drawn first and objects closest to the "top" are drawn last. The order in which graphic objects occur in collections implicitly specifies this sequence.

This ordering is necessary because the two-dimensional display has no other way of knowing which of two (or more) overlapping graphic images should be visible. In two-and-a-half-dimensional graphics, in which the graphic objects are planar and parallel to the viewing surface, but have explicit z-depths, the implicit ordering of a collection is not necessary. Objects may be produced in any order and the rendering software sorts them into appropriate layers by their z values. Similarly with three-dimensional graphics, the rendering software determines which graphic objects are visible.

The upcoming sections describe the different types of graphic group.

3.4.1. Graphic collection

The graphic collection object is the simplest of the grouping operators; it merely allows several graphic objects to be treated as a single unit. It supplies a spatial transformation to all of its children. This transformation temporarily modifies the existing coordinate system. It only applies to the children of the group; it is removed after they have been drawn.

	Collection	**Replication**	**Selection**	**Domain**
Spatial Attribute	originX originY scaleX scaleY rotate	originX originY scaleX scaleY rotate	originX originY scaleX scaleY rotate	Float Float Float Float Float
Group Attributes		copies index	 choice	Integer Integer Integer

Table 5 Types of graphic groups

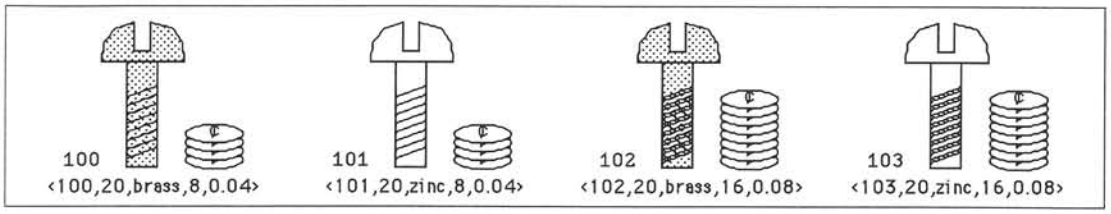

Figure 5 Graphic relation with replications

3.4.2. Graphic replication

The graphic replication object draws its components zero or more times depending on the number of copies specified. For example, in the graphic schema of Figure 4, the ellipse that will form the stack of coins occurs within a replication. The number of copies is set to a function of the cost. The expression converts the fractional dollar value of the bolts into the number of coins: *stack.copies := floor(cost * 100)*. The name of the replication object is *stack* and the *floor* function converts a floating point parameter to an integer result.

The index attribute identifies each individual copy. The index starts at 1 and stops at the number of copies. Subobjects within the replication may bind to this attribute to distinguish each copy. The index may be used as an input in a binding expression on any of the attributes of the children (or grandchildren, etc). In the case of the coin stack, the vertical position of the ellipse, relative to the group it is in, is given by the expression: *coin.bottom := bolt.bottom – 10 * stack.index*. The name of the ellipse making the coin is *coin*. The minus sign comes from the display origin being in the upper left corner—the stack of coins grows upwards.

The index is the only kind of local variable that the expression language provides. The information it provides distinguishes each replication. If none of the graphic objects bind to the index, then they will all be drawn identically on top of each other. The graphic objects of each higher index number are drawn on top of those of lower index numbers. The copy with value 1 is drawn first, and later copies are drawn on top of it. Care must be taken to ensure that the generated images appear correctly layered. For the coin stack example of Figure 4, the bottom coins must be drawn first so that they will be below those on top.

The replication group can be used to specify useful graphical structures. For example, in the bolts example (Figure 3), the number of turns per inch can be visualised by actually drawing

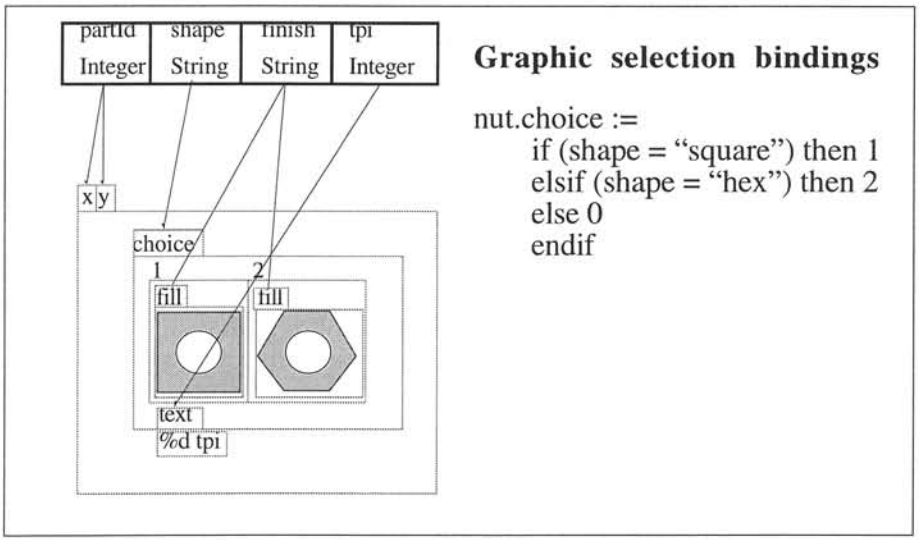

Figure 6 Nut relation graphic schema

the turns onto the bolt. A line segment appears in the graphic schema in Figure 4 in order to represent the turns—it occurs within a replication for which the number of replications is *tpi*. Although the binding could be similar to that in the coins example above, it would look better if the turns stayed within the length of the bolt. The shortest bolt is always 20 in length and the turns can be evenly spaced over the last portion of the bolt by relating the position of the turns to the total size. The *turns* object is the replication containing the *thread* object that is the line segment. The *bolt* object is the rectangle that represents the body of the bolt.

turns.copies := tpi
*thread.top := bolt.bottom – turns.index * (20 / tpi) – 4*
*thread.bottom := bolt.bottom – turns.index * (20 / tpi)*

In the bindings, *bolt.bottom* refers to the bottom edge of the bolt. Each turn is drawn a certain amount upward from the bottom and the distance between the turns is derived from to the turn density. In other words, the greater the number of turns, the more densely they appear on the image. Figure 5 shows a partial result from the relation. Four tuples have been instantiated with bolt information. The source data is placed below each bolt and is not part of the graphic relation.

3.4.3. Graphic selection

A graphic selection draws only one of its members, chosen by an integer index. Every subobject of a selection has a unique identifier in the range of 1 to the number of objects. The object whose number matches the choice attribute is selected. If the choice is less than 1, or greater than the number of subobjects, nothing is drawn.

A graphic schema and bindings for the *Nut* relation are shown in Figure 6. The schema contains a selection object with two choices. A conditional expression returns 1 if the nut shape is the string *square* or 2 if it is *hex*. The selection object will draw the square nut shape when the value is 1, or it will draw the hex polygon shape when the value is 2. The square nut is a filled rectangle with a white-filled circle on top of it. The hex nut is a filled polygon topped with a white-filled circle.

4. Relational Visualisation Notation

The Relational Visualisation Notation is a graphical language for specifying operational and reusable visualisation designs. It depicts data relations, graphic relation designs and relational transformations with graphical symbols. These symbols can be interconnected to form visualisation designs which, when interpreted, produce visualisations. Visualisations are compositions of multiple, interrelated graphic relations. Design fragments can be stored in a library and referenced by other designs; they support prototyping, development and evaluation within the visualisation lifecycle.

Essential symbols in the notation refer to database relations and relational operations such as select, project and join. These transformational symbols combine and relate the data and graphic relations. Other symbols represent new types of graphic relations: transformations and sources. Transformations generate informational output that carries graphic information. Graphic sources generate informational output solely from their visual appearance; they have no input. Special symbols represent subdesigns, and their input and output parameters.

Designs are collections of symbols that are connected with arrows into directed, acyclic graphs. Arrows denote the flow of data from relations and graphic sources, through operators and transformational graphic relations to graphic outputs. In this way, designs describe the relationships among a set of relations and a visual display. All types of graphic relation generate visual output: source, transformational and output.

Section 4.1 formally defines the notation's relational operators and provides examples of their behaviour. Section 4.2 defines Relational Visualisation Notation designs to be graphs of operators and presents several simple design scenarios. Graphic relations, originally described as outputs in Section 3, are extended in Sections 4.3 and 4.4 to become graphic sources and graphic transformations. Section 4.5 defines additional notational constructs for specifying reusable visualisation designs. A sample design, some data and their corresponding visualisation appear in Section 4.6.

4.1. RELATIONAL ALGEBRA

Relational algebra is set of formal composition operators applied to relations. Each operator maps one or more input relations to an output relation. Operators themselves may be composed to construct more powerful transformations. There is no canonical specification of relational algebra: it has been formulated differently in many query languages [Hawryszkiewycz 84] [Zloof 75] [Poulovassilis 92]. It has been proved equivalent to other notations, such as relational calculus and domain calculus [Ullman 83], and any implementation that contains a basis set of operators will be functionally equivalent to it.

For precision and to avoid confusion, this research explicitly defines its own set of relational operators. The basis set consists of SELECT, PROJECT, MAP, JOIN, DIVIDE, UNION, INTERSECTION and DIFFERENCE operators. The first three compute a new relation from an existing one. The next two combine two relations to produce a third and the last three manipulate relations as sets. Extra operators PAIR, EQUIJOIN and SYMMETRIC-DIFFERENCE are included for convenience. All operators produce new relations for output; they do not modify their inputs. Furthermore, the schema of each output relation is statically derivable from the schema of the input relations and the formal description of the operator.

Table 6 defines formal nomenclature for describing the relational operators. Recall from Section 2.3 that a relation is a subset of the cross-product of some domains. D and E refer to domains, such as Integers, Floats or Booleans, each of which is a set. The relations are indicated by R, R_1 and R_2. Juxtaposing two tuples represents their concatenation. The elements of the second tuple immediately follow the elements of the first. The definition of equivalence classes is used by the SEQUENCE and SUMMARISE operators to specify subgroups. Each subgroup is an equivalence class in which, for all of the attributes in list G, all of the tuples within each class have the same values for those attributes and any tuples in separate classes have at least one differing value.

Table 7 formally defines the chosen operators and provides a brief description of each. Each operator has relational inputs, denoted by R, and naturally generates one output, denoted by S. Most operators require additional design information to specify their precise semantics—they may require a list of attribute names, or a function in order to be fully specified.

D, E	attribute domain set, such as Numbers, Booleans, Strings, etc.
R, R_1, R_2	input relation $R \subseteq (D_1 \times D_2 \times \ldots \times D_k)$
uv	tuple concatenation such that $u \in (D_1 \times \ldots \times D_k)$, $v \in (E_1 \times \ldots \times E_j)$ $(uv)_i = (i \leq k, u_i), (i > k, v_{i-k})$
G	$a_1 \ldots a_g$, $g \geq 0$, $1 \leq a \leq k$, sequence of attribute identifiers used to specify equivalence classes
$[R]_G$	Set of equivalence classes of R, partitioned by G $\{S \mid S \subseteq R \wedge (\forall_{s,t},\ s, t \in S, \forall_b, 1 \leq b \leq g, s_b = t_b)$ $\wedge (\forall_{s,t},\ t \in R, s \in S, (\forall_b, 1 \leq b \leq g, s_b = t_b) \rightarrow (t \in S))\}$

Table 6 Nomenclature

Operator	Function
SELECT(R,bf)	Output those tuples in R for which the tuple function bf is true $R \subseteq D_1 \times \ldots \times D_k$, bf:$D_1 \times \ldots \times D_k \rightarrow B$ SELECT(R,bf) = $\{t \mid t \in R \wedge bf(t)\}$
PROJECT(R, l)	Output all tuples with only the attributes indexed in the list, and eliminate duplicate tuples. $R \subseteq D_1 \times \ldots \times D_k$, $l = a_1, a_2, \ldots, a_q$; $\forall_i$, $1 \leq i \leq q$, $1 \leq a_i \leq k$ PROJECT(R, l) = $\{(ta_1, ta_2, \ldots, ta_q) \mid t \in R\}$
PAIR(R_1, R_2)	Output the cross product of the two relations $R_1 \subseteq D_1 \times \ldots \times D_k$, $R_2 \subseteq E_1 \times \ldots \times E_j$ PAIR(R_1, R_2) = $\{uv \mid u \in R_1, v \in R_2\}$
JOIN(R_1, R_2, bf)	Select from the cross product of R_1 and R_2 those tuples where bf(t_1t_2) is true $R_1 \subseteq D_1 \times \ldots \times D_k$, $R_2 \subseteq E_1 \times \ldots \times E_j$, bf: $(D_1 \times \ldots \times D_k) \times (E_1 \times \ldots \times E_j) \rightarrow B$ JOIN(R_1, R_2, bf) = $\{uv \mid u \in R_1, v \in R_2, bf(uv)\}$
EQUIJOIN (R_1, R_2, p)	Select from the cross product of R_1 and R_2, those tuples with corresponding values in attributes indicated in the pair list $R_1 \subseteq D_1 \times \ldots \times D_k$, $R_2 \subseteq E_1 \times \ldots \times E_j$,p = $((d_1,e_1), .., (d_q,e_q))$, $\forall_i$, $1 \leq i \leq q$, $1 \leq d_i \leq k$, $1 \leq e_i \leq j$, $D_{di}=E_{ei}$ $e(u,v)=(u_{d1},..,u_{dq})(u_m, \forall_m\, 1 \leq m \leq k, m \notin p_d)(v_m, \forall_m 1 \leq m \leq j, m \notin p_e)$ EQUIJOIN (R_1, R_2, p) = $\{e(u,v) \mid u \in R_1, v \in R_2 \wedge (\forall_i, 1 \leq i \leq q, u_{di} = v_{ei})\}$
MAP(R, f)	Map each tuple in R through function f to a new tuple in the output $R \subseteq D_1 \times \ldots \times D_k$, f: $D_1 \times \ldots \times D_k \rightarrow E_1 \times \ldots \times E_j$ MAP(R, f) = $\{f(t) \mid t \in R\}$
SUMMARISE (R, G, I, sf)	For each equivalence class in R, output a single summation tuple $R \subseteq D_1 \times \ldots \times D_k$, $I \in E_1 \times \ldots \times E_j$ sf: $(D_1 \times .. \times D_k) \times (E_1 \times \ldots \times E_j) \rightarrow E_1 \times .. \times E_j$ SUMMARISE (R, G, I, sf) = $\{z_{hg} \mid \forall_h, h \in [R]_G, z_{h0} = I, \forall_t, t \in h,$ $z_{hi} = sf(t, z_{h(i-1)}), g = \lvert h \rvert\}$
SEQUENCE (R, G, T,f_1, f_n)	Map each tuple of R, in sorted order, to a new tuple, using the tuple, the previous tuple and the previous output as input $R \subseteq D_1 \times \ldots \times D_k$, T: $(D_1 \times \ldots \times D_k) \times (D_1 \times \ldots \times D_k) \rightarrow B$ f_1: $D_1 \times \ldots \times D_k \rightarrow E_1 \times \ldots \times E_j$ f_n: $(D_1 \times .. \times D_k) \times (D_1 \times \ldots \times D_k) \times (E_1 \times \ldots \times E_j) \rightarrow E_1 \times .. \times E_j$ SEQUENCE (R, G, T,f_1, f_n) = $\{z_{hi} \mid \forall_h, h \in [R]_G, A_t, t \in h, (\sim\exists s, s \in h,$ $T(s,t))\ z_{h1} = f_1(t), (\exists s, s \in h, T(s,t), \sim\exists v, v \in h, T(s,v), T(v,t))\ z_{hi} = f_n(t, s, z_{hi-1})\}$
UNION (R_1, R_2)	Set union, $R_1, R_2 \subseteq D_1 \times D_2 \times \ldots \times D_k$ $R_1 \cup R_2 = \{t \mid t \in R_1 \vee t \in R_2\}$
INTERSECTION (R_1, R_2)	Set intersection, $R_1, R_2 \subseteq D_1 \times D_2 \times \ldots \times D_k$ $R_1 \cap R_2 = \{t \mid t \in R_1 \wedge t \in R_2\}$
DIFFERENCE (R_1, R_2)	Set subtraction, $R_1, R_2 \subseteq D_1 \times D_2 \times \ldots \times D_k$ $R_1 - R_2 = \{t \mid t \in R_1 \wedge t \notin R_2\}$
SYMMETRIC-DIFFERENCE (R_1, R_2)	Set exclusive-or, $R_1, R_2 \subseteq D_1 \times D_2 \times \ldots \times D_k$ $R_1 \otimes R_2 = (R_1 \cup R_2) - (R_1 \cap R_2)$
DIVIDE(R_1, R_2, p)	Inverse equijoin of the two relations $R_1 \subseteq D_1 \times \ldots \times D_k$, $R_2 \subseteq E_1 \times \ldots \times E_j$,p = $((d_1,e_1), .., (d_q,e_q))$, $\forall_i$, $1 \leq i \leq q$, $1 \leq d_i \leq k$, $1 \leq e_i \leq j$, $D_{di}=E_{ei}$ r = q – | p |, f = $b_1, \ldots, b_r$, $\forall_i$ $1 \leq i \leq r$, $b_r \notin p_d \wedge \forall_{g,h}$, $1 \leq g,h \leq r$, $g < h \rightarrow b_g < b_h$, Q = $F_1 \times \ldots \times F_r$, $\forall_i$, $1 \leq i \leq r$, $F_i = D_{bi}$ $s(u,v) = s_1,\ldots s_k$, $1 \leq i \leq k$, $(i \in p_d) \rightarrow (s_i = u_g \ni p_{gd} = i)$, $(i \notin p_d) \rightarrow (s_i = v_g, b_g = i)$ DIVIDE(R_1, R_2, p) = $\{v \mid v \in Q, \forall_u \in R_2, s(u,v) \in R_1\}$

Table 7 Summary of relational operators

The SELECT operator creates a new relation by selecting from the input all those tuples for which the Boolean function evaluates to true. The function takes one tuple for input and returns a Boolean that indicates whether or not the tuple should appear in the output. The PROJECT operator copies each tuple of the input relation to the output, eliminating those attributes not in the given list. Because eliminating attributes may cause tuples to become identical, PROJECT preserves tuple uniqueness by removing duplicates. The UNION, INTERSECT and DIFFERENCE operators are the same as the corresponding set operators where relations are treated only as sets with tuple elements. The SYMMETRIC DIFFERENCE operator is the exclusive-or set operator.

Three kinds of join operator are provided, each using two input relations called "left" and "right". The EQUIJOIN operator creates new tuples by taking combinations of tuples from its inputs for which the values of specified attributes match exactly. The specified attributes must occur in both inputs and must have the same domain. The schema of the result is generated by first listing each one of the attribute pairs that are used for matching, since these are always common to both relations. This is followed by those remaining attributes of the first relation (on the left) and then the remaining attributes of the second relation (on the right). Attribute names that are common to both input relations, but that are not used for matching, are prefixed with "left_" or "right_" to distinguish them.

A general-purpose JOIN selects tuples from the cross product of the inputs, using an arbitrary Boolean function to determine if the result belongs in the output. The function accepts a pair of tuples for input—one from each relation. It returns true if the pair belong in the output, false otherwise. The PAIR operator generates the complete cross product of the two input relations. It is often used to associate common information with all of the tuples of a relation. It can also be used to construct two-parameter functions from one-parameter operators. Notice that JOIN can be constructed by taking a PAIR and a SELECT. Conversely, a PAIR is a JOIN with a constant true function.

The MAP operator creates one output tuple for every input tuple, computing the output as a multi-valued function of the input. Each instance of the MAP operator has an input schema that specifies the structure of tuples that it accepts and an output schema that defines the tuples that it produces. Each output attribute is computed from an arithmetic expression. The output attributes are all computed independently.

The DIVIDE operator is rarely used and provided only for the completeness of the basis set. Relation *r1* is the dividend into which the divisor *r2* is to be divided across the attributes given in the list. The attributes of the quotient are the attributes of the dividend less those in the division list. A tuple appears in the quotient if all the pairs it makes with the divisor appear in the dividend. That is, the quotient, when EQUIJOINed with the divisor across the attribute list, produces a subset of the dividend. Every item of the quotient must exist in the dividend in all combinations with those items in the divisor [Hawryszkiewycz 88].

Two special relational operators increase the power and flexibility of the notation. The SUMMARISE operator accepts a relation as input and generates a single summary tuple for output. It computes maxima, minima, totals or other values that can be computed in one pass through the input tuples. A similar operator appears in the Iconographer system [Draper 90].

The SEQUENCE operator maps an input relation to an output relation by sequentially processing tuples in the order specified by a comparison function. The output tuples are computed from expressions involving data from both the previous output tuple and the previous input tuple. The operator can model sorted lists by using the comparison function to sort the input tuples, and then numbering the output tuples 0, 1, 2, etc. Using a similar technique it can also compute partial sums, such as the successive positions of wedges on a pie chart. It is provided to enable strictly set-based relations to model lists, whereas commercial databases allow relations to vacillate between sets and lists.

Both of these new operators can be applied to groups—equivalence classes or disjoint subrelations within the input relation. The operator applies to a group independently of other groups. The results of the operation on each disjoint group are then unioned to produce the output. These operators do not rely on programmed loops or recursion, and they maintain the definition of relations as sets, rather than lists.

4.2. DESIGN GRAPHS

Graphical symbols in the notation denote data relations, graphic relations and relational operators. Data relations are represented by rectangles that enclose the name of the relation. Rectangles with thickened ends enclose the name of their graphic relation. Ovals enclose the name of their operation: PROJECT, SELECT, etc..

Small black squares attached to the symbols are piers to which arrows can connect. Piers on top of the symbol are for input and those below are for output. While output piers are annotated with the schema of the relation that they produce, input piers are not. As long as the diagram is well-formed it is always possible to derive the output schema of every node.

A node may not connect to itself, and arrows must not form cycles. Data may not flow out of one node and into another that returns it back to the source. Disallowing cyclic flow graphs does not weaken the notation, because cycles provide few benefits and are difficult to compute. Other design systems also do not permit them [Myers 87] [Barth 86] for similar reasons.

From the design graph alone, the output schema of each operation is derivable from its input schema and its own specification. Because a schema can be determined for every output pier, the design can be checked for well-formedness statically and automatically. A design is well-formed if every input pier has exactly one input connection, there are no cycles and all arithmetic expressions are well-formed. The first two conditions are easily checked by examining the graph connectivity and the third by parsing the expressions. Expression parsing must occur in the order of node dependencies because the success or failure of any node determines the acceptibility of any node further down the line.

Figure 7 shows the three basic types of symbols in the notation. Figure 7a is the data relation *Salaries* with its schema. Figure 7b shows the graphic relation *Salary-age plot.* Figure 7c shows a JOIN operation that selects the tuples for which attribute *a* matches attribute *b*. Additional operational design information, such as functional components or attribute names, appear adjacent to the node.

Designs are created by connecting output piers to input piers with arrows. An output pier may have any number of arrows leading away from it, and exactly the same data are input everywhere that the arrows connect. In contrast, an input must have exactly one edge entering it. An input with no connecting edge has no data and an input with two edges is not permitted.

Figure 8 shows a simple design diagram. The source symbol contains the name of the Salaries relation, which is to be visualised. It generates one relation for output but requires no input. The graphic relation is a data sink that has input, but no informational output. No other transformations are used in this figure and the data flows directly from the source to the graphic

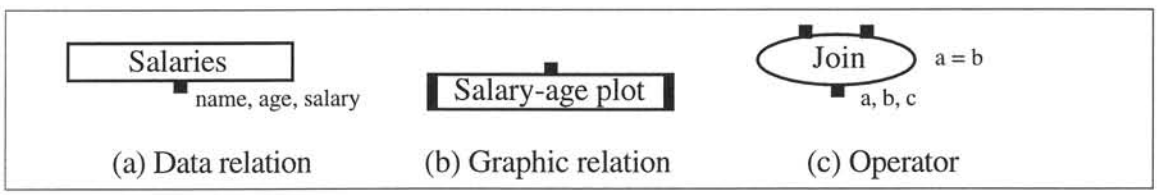

Figure 7 Basic graphical symbols

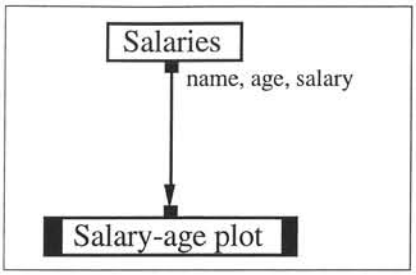

Figure 8 Simple design relation.

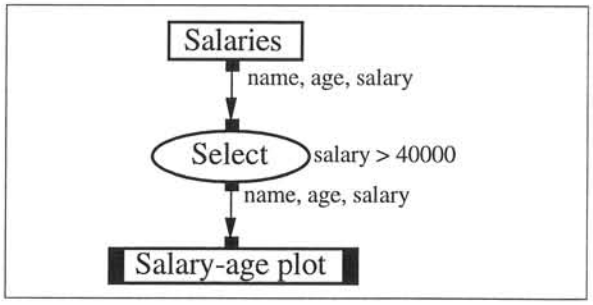

Figure 9 Design with selection operation

Normally data must be manipulated to produce a visualisation, as in the example in Figure 9. Operation ovals have one or two inputs and one output. Parametric design information appears to the right of the oval. Projections require a list of the attribute names to project onto; selections require the selection condition. In the example, a SELECT first determines which tuples from the relation should be displayed and the chosen data then appears in a graphic relation.

The JOIN, EQUIJOIN and PAIR operators take two relations for input and generate a new relation for output, as illustrated in Figure 10. The *Projects* relation shows which people are working on which projects. The *Priorities* relation indicates the relative importance of each project. The result shows the priority of the project that each person is working on, visualised as a project priority graph. The graph is produced by a subdesign, which is represented by the double-line box labelled *Project priority plot*. Subdesign creation and usage is discussed in Section 4.5.

A design may have several graphic relations showing data in different ways. In Figure 11, the Priorities relation produces two different displays. One shows the data as a pie chart and the other shows it in combination with the Projects data. Both displays are produced from subdesigns. In addition, the design uses the summarise operator to determine the number of people working on each project. It counts the number of tuples associated with each project number.

4.3. GRAPHIC TRANSFORMATIONS

Although graphic relations are suitable for displaying relational information, the graphical information they produce while doing so is not available to other graphic relations. Information such as the positions of graphic objects, angles or distances between graphic objects are useful

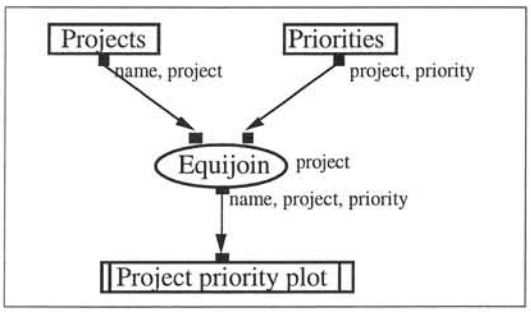

Figure 10 Two relations combine to produce one output

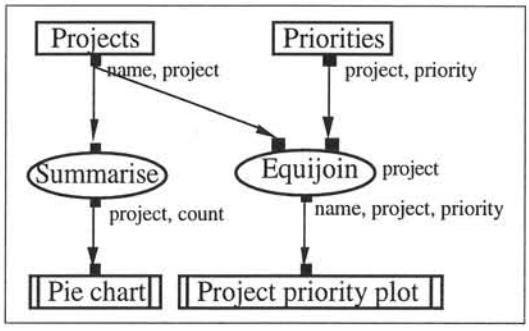

Figure 11 Design with two relations and two outputs

in coordinating the graphics of several graphic relations. In addition, it would tedious to have to define all information relationally first. Some information is inherently graphical, such as the location and sizes of axes or positions on a background map. To solve these problems we define two new types of graphic relations. A *graphic transformation* both renders an image and outputs relational information. It is similar to a MAP operator, except that the expressions involve the attributes of graphic objects. A *graphic source* also renders an image and generates information, but it has no relational input. An annotated picture with a relational equivalent, it allows graphical information to be specified graphically. Graphic sources are described in Section 4.4.

Graphic transformations extend a graphic relation with an output relational model. Like the input model, it has a series of attributes, each distinguished by a name and a domain. Each output has an associated arithmetic expression that maps from the input attributes and object attributes to a single value.

For example, a graphic transformation may map hours to locations on a clock, as in Figure 12 below. Figure 12a shows an input schema for appointment information—hourly entries have associated messages. The graphic schema, consisting of a clock face and hour hand, appears in Figure 12b. Each string will be placed at the hour-position on a clock, with other information, during a subsequent graphic relation. The hour hand object is named *hand* and the clock face is named *face*. A line of thickness 0 (and hence not visible) called *spot* marks the position that the message will be placed at. It is the same as hand, but longer.

Figure 12c shows the binding functions of the graphic schema. The length of the spot line is just longer than the radius of the clock face. The angle, in degrees, of the hour hand is computed from the hour, irrespective of the precise location of the hand. The output bindings in Figure 12d compute the message position and direction from the end point of the hand. The *up* and *left* attributes indicate how the message is to be placed relative to the coordinates. Up indicates that the message should appear above the point. Left indicates that the start of the message is to the left of the point. False values indicate the reverse. The attributes of the hand are computed automatically when the graphic relation is rendered.

The clock hands graphic relation would be used in a design such as in Figure 13. Data from a schedule describe the hour of an appointment with an accompanying message. They do not necessarily come directly from a source relation, but could be derived from a calendar joined with the current time, or other means. The hour data are converted to x, y positions of where

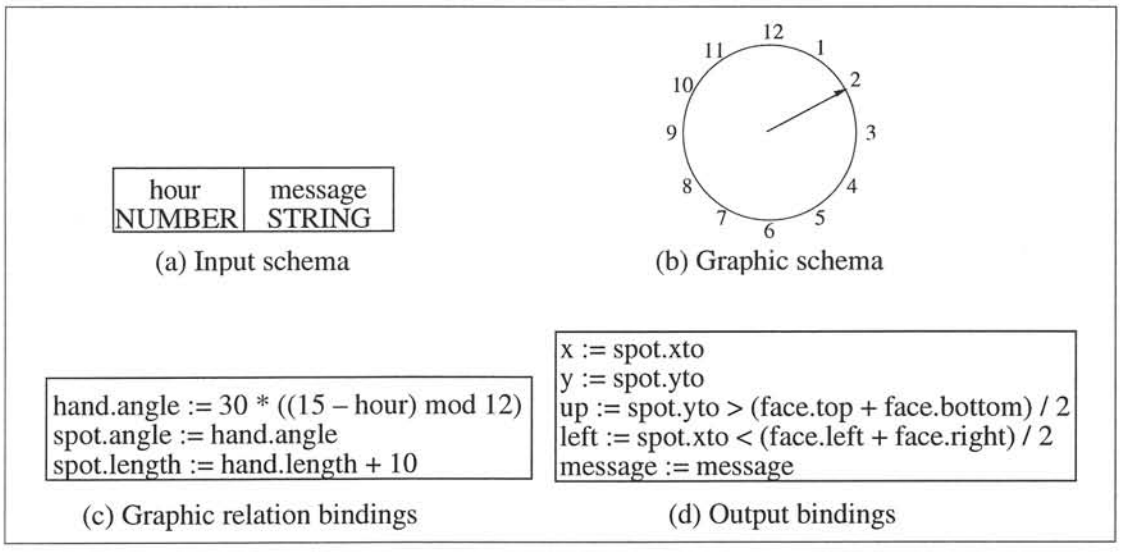

Figure 12 Circle of clock times with bindings

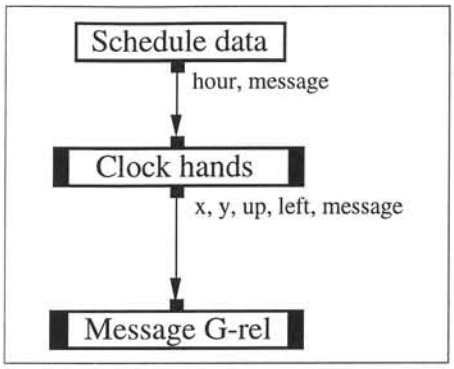

Figure 13 Schedule design

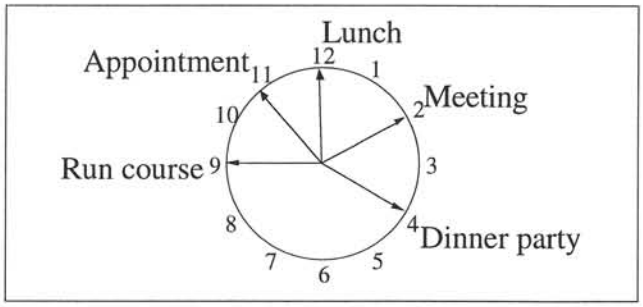

Figure 14 Resulting visualisation

the message should be located. The generated x and y positions come from the end point of the clock hand. The up and left values arrange the text next to the clock so that the text does not overwrite the clock itself, but abuts it neatly. The message is copied through directly. Both graphic relations are rendered together and the result appears in Figure 14.

4.4. GRAPHIC SOURCES

Unlike graphic relations, a graphic source does not have an input schema, yet it has an output schema for the information that it generates. The source contains graphic objects—boxes, circles, etc.—just like an ordinary picture, as well as *figure tuples*. A figure tuple is a special kind of group of graphic objects that designates an information tuple. A graphic source produces one output tuple for every figure tuple. Each figure tuple contains its own bindings for the output attributes—when they are evaluated they produce distinct output data tuples.

Although a figure tuple is like a group, there are several important distinctions. The tuples appear only in graphic sources. They do not form hierarchies—rather, they are always the immediate children of the root node and may not contain other figure tuples. Figure tuples define their own name space—the names of their graphic objects are local within them. It is common for distinct figure tuples to have duplicate object names because they use them to vary the output data.

Graphic sources generate relational data one graphic tuple at a time. Each figure tuple evaluates its output bindings in its own name space. The names used in the expressions may refer either to global objects (those not in any figure tuple), or local ones. The values of the globally-named objects are the same for every figure tuple and the locally named ones are different. By this mechanism, each figure tuple can produce a distinct data tuple whose values depend on the figure tuple's graphical elements. Section 5.2 contains an example of the steps a user might take to create a graphic source.

Figure 15 shows the information that would be used in a legend that relates the finish on

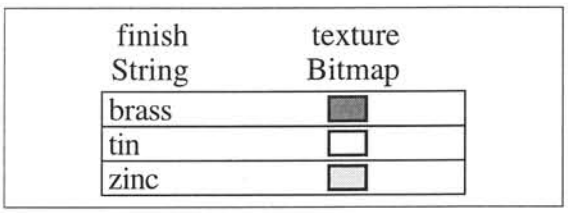

Figure 15 Output information model

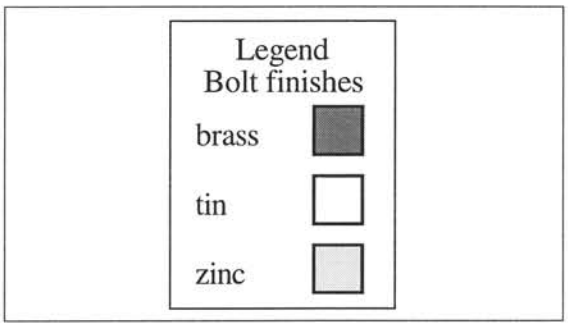

Figure 16 Graphic source legend

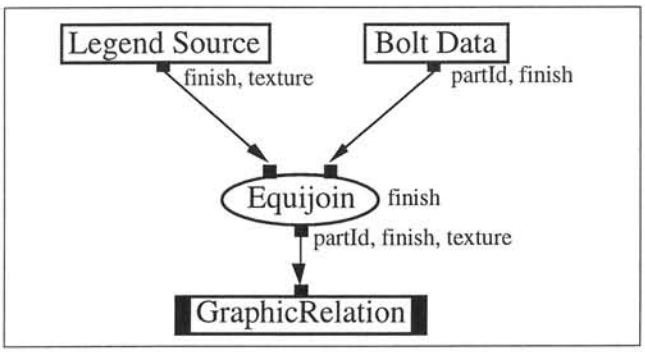

Figure 17 Composing a graphic source with data for an output relation

bolts or nuts to particular textures. The finish is a text string that matches those used in the nuts and bolts relations from the example in Section 3. The texture is the bitmap for filling the figure. Rather than filling out a relation with the information, it is much more appropriate to simply draw the legend, which appears in Figure 16. Each rectangle and name pair is a figure tuple. The graphic source output attributes are bound to the text field's contents and the rectangle's fill pattern. Rendering the picture generates the corresponding data.

The legend in Figure 16 can be composed with the bolt data from Section 3 to provide the texture patterns necessary for the display. The design diagram and a result of the relations are given in Figure 17. The EQUIJOIN translates the name of the finish on each bolt into the specific texture pattern. Each bolt is then drawn with the appropriate shading. Furthermore, changing the picture of the legend automatically changes any derived displays.

4.5. DESIGN REUSABILITY AND ABSTRACTION

One of the goals of visualisation design is to produce a library of useful visualisations. These visualisations can be tested over a long period of time, empirically evaluated, refined and distributed to others. Novel designs appearing in some projects may be abstracted and placed into a library without having to redesign and re-test them. Frequently used displays, such as lists, maps, charts and iconic displays can be made available for rapid prototyping. Reusable designs support the long term development, prototyping and evaluation of visualisations.

Two complementary mechanisms allow designs to be reused. Parameterisation allows designs to have formal argument relations rather than actual database relations. The parameters are instantiated when the design is activated. Abstraction allows a design to be used as a component within another design. The subdesign is treated as a single unit, as if it were a graphic relation that visualised several input relations, rather than just one.

4.5.1 Parameterisation

Designs may have zero or more input parameters and zero or more output parameters. Named, parameter relations are defined by a schema, but contain no data. Their data come from the design that uses the subdesign and appear only when the design is interpreted. Parameters have unique names and sequential index numbers. The names are mnemonic, but the numbers denote the input and output piers of the design node. For example, input parameter 1 is the leftmost input pier. The input and output parameters are numbered in a separate sequence.

Figure 18 shows the definition of a subdesign for the pie chart. The input parameter is a special source relation called *Pie data #1*. The bulk of the design generates the pie wedge sizes from the data and sums their values (with the sequence operator) in order to determine the starting position of each wedge. Each pie wedge is annotated with the midpoint of its arc and two Boolean flags that indicate if the midpoint is above the centre point and/or to the right of the

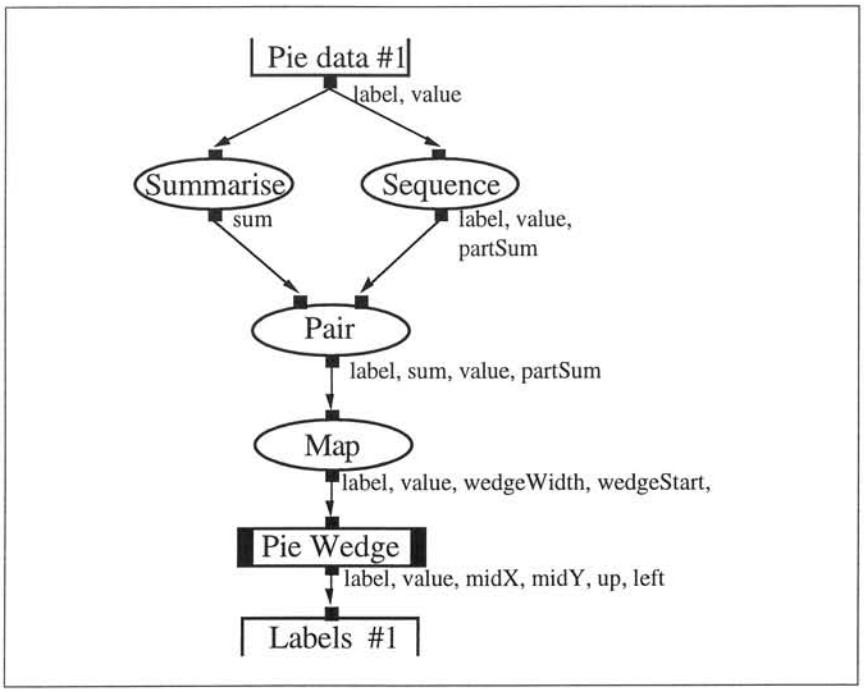

Figure 18 Definition of a reusable pie chart

centre point. This information may be used in later stages to associate further information with the pie wedges. The resulting data is sent to output parameter *Labels #1*.

Parameterised designs may still reference named relations from the database. Using named relations fixes the design to always work with a particular database relation. In fact, nonparameterised designs are a special case; they contain no input parameters. A design with parameters may be interpreted on-the-fly by dynamically specifying the relations that should be applied to the input parameters.

4.5.2 Abstraction

A design used from within another design behaves like a complex graphic relation. The subdesign has inputs to which arrows must be attached, and it may have outputs that produce graphical information that is useful elsewhere. It is rendered as a single image, even though it will be made of several graphic relations. A subdesign may be used more than once within several designs, or even several times within the same design. The notation denotes the use of a subdesign by a double-barred box containing the design's name.

Figure 19 shows a design that uses the subdesign for the pie chart from Figure 18. The pie chart subdesign accepts a relation that contains value-name pairs. The pie chart also generates output data, but this is not used, and so the figure is not further connected.

Subdesign parameter relations are bound by position. The input and output piers are each intrinsically numbered from left to right, starting at 1. The numbers correspond to the design's relational parameters. Parameters are compatible if the domains of their schema match. Positional matching is used both for the parameters and their attributes because it improves design reusability. The input attributes are unlikely to have the same names, but they can easily be arranged to have the same positions. Furthermore, it is the same technique used for graphic relations.

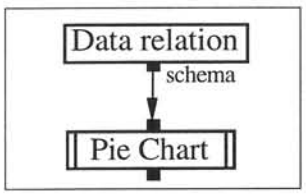

Figure 19 Use of a reusable design

troops Integer	**fromPlace** String	**fromDate** String	**advanced** Boolean	**losing** Integer	**toPlace** String
422000	Kawno	?	true	0	Kawno East
98000	Tarantino	18 Oct	false	2000	Malejaies
50000	Botr	21 Nov	false	22000	Studienska

Table 8 Schema of troop movement data

The graphic relations and subdesigns used within a design are rendered in an order specified by the user. Rendering graphic relations produces the image associated with the its data relation. Subdesigns are rendered recursively. Their component graphic relations and subdesigns are rendered as if the main subdesign were a design itself.

4.6. SAMPLE DESIGN AND VISUALISATION

date string	**temp** float
14 Nov	–21.0
6 Dec	–30.0

Table 9 Temperature schema

This example shows a design for Minard's visualisation of Napoleon's 1812 campaign against Russia, originally introduced in Section 2. Minard's graphic is based on data that describe the movement of troops between cities. Each tuple describes a collection of soldiers who travelled from one place to another on a particular date. In the schema of Table 8, *troops* is the number of soldiers, *fromPlace* and *toPlace* the origin and destination city names, *fromDate* the date the troops left the city, *advanced* is true if the troops were advancing and false if they were retreating, and *losing* is the number of soldiers that perished along the way. The data for the relation are taken from Minard's hand-drawn picture; Table 8 shows three of the tuples.

Data have also been extracted from Minard's graphic to show the temperature on particular

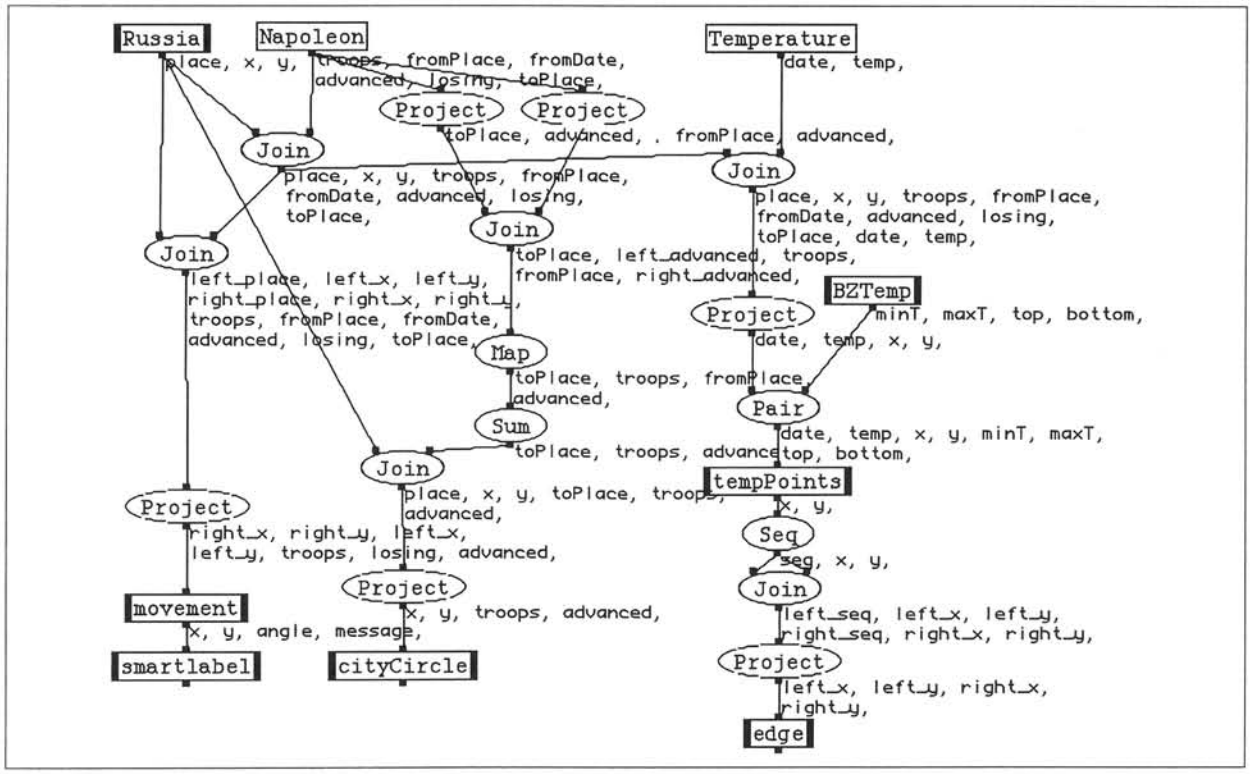

Figure 20 Visualisation design

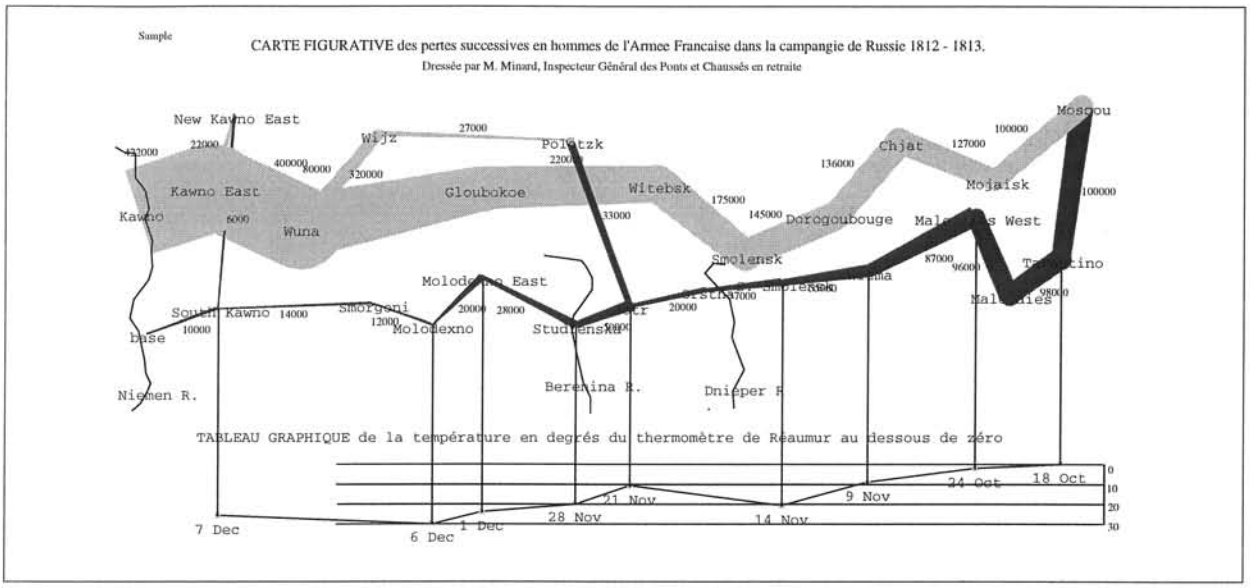

Figure 21 Reconstruction of Minard's graphic

days. In the schema of Table 9, *date* gives the date on which the temperature was recorded, while *temp* holds the temperature in degrees of Réaumur's thermometer.

The visualisation design appears in Figure 20; it has three main parts. The left third of the diagram represents troop movements as the thickness of the line segment. The middle third rounds off the angular connections of troop lines at cities. The right third plots the temperature on a line graph below the main visualisation and relates the data points to cities. In addition to the two data relations previously described, the design uses a graphic source to provide initial data. The graphic source contains a map of Russia and generates the names of cities and their locations as output.

The troop movements are computed in the left third by joining the data twice to the map of Russia. These joins, followed by a project, convert the names of the origin and destination cities into coordinates. These, along with the troop sizes, enter the movement graphic relation which renders them as a polygon whose ends are perpendicular to the direction of travel and proportional to the troop departure and arrival populations. The *smartlabel* graphic relation that follows the movement graphic relation places the troop size on the midpoint of the troop polygon, in such a way that the text is not obscured by the polygon.

The troop movement path created from the polygon segments described above has angular joints over the cities. Filled circles can be added on top of these joints to smooth them out. The affected joints are determined by selecting all the cities which have a troop path coming in and a troop path going out—thus discarding end-points. For the joint cities, the troops will be considered to be advancing if either of the troop paths are advancing. The advancing trait is needed in order to make the circle fill patterns match those of the path. The map operator computes the correct advancing value. The coordinates of the cities are then determined from the map of Russia by joining with it. The resulting city circles are rendered with the *CityCircle* graphic relation.

The temperature plot is largely independent of the troop movement path. However, it does need to determine the city the troops were at when the temperature was recorded, the coordinates of the city on the map, and whether or not the troops were advancing or retreating. The data are determined by joining the temperature data with the troop movements. During the join, only retreating movements are considered because the temperature plot only records retreats. The projection after the join produces the temperatures on selected days of troop

movements during the retreat and the coordinates of the city. The data are paired with data describing the temperature scale, which is another graphic source. The resulting data are plotted as points, vertical city-connection lines (but not between-point lines), and dates along the line graph. The plots of the points, however, are then sequenced by date and joined each to the next sequential date, producing pairs of points. These pairs are graphed as the line graph.

The PostScript output of the Relational Visualisation Toolkit appears in Figure 21. City names and troop sizes appear on top and the troop movements appear as a continuous path beneath them. The line graph correctly indexes the cities.

5. Relational Visualisation Toolkit

For the Relational Visualisation Notation to support the visualisation prototyping process effectively, the notation itself must be supported by tools. Interactive direct-manipulation tools allow people to create, edit and test designs. The notation can be shown to be accessible by describing the Relational Visualisation Toolkit, a set of direct-manipulation tools, and by showing how they may be used to create a realistic visualisation design.

The Relational Visualisation Toolkit consists of four major tools and a number of subtools, as shown in Table 10. The major tools are the *System Browser*, the *Relation Browser*, the *Graphic Relation Browser* and the *Design Browser*. The System Browser opens the door to the toolkit by allowing the user to create, organise and prototype the three main notational components: relations, graphic relations and designs. The Relation Browser provides access to data relations. The Graphic Relation Browser supports the creation of graphic relations and graphic sources, and includes two subtools for creating interobject and output bindings. The Design Browser allows the user to create, edit and prototype visualisation designs. In addition, every kind of relational operator in a design has its own editor for specifying the operator's parameters, and the Tile Browser allows the user to specify the tiles of a design and the layers within those tiles.

These tools have been implemented in Smalltalk/V on a Macintosh PowerBook 520. The user interface software consists of object-oriented classes, each providing a different interactive interface. In addition, much of the code for producing the graphics and responding to events is associated with the application classes directly. The user interface accounts for 40% of all of the code in the system, about 12,000 lines of Smalltalk spread over 37 classes.

The toolkit is freely available as a Smalltalk/V Mac image and source code at

Tool – subtool	**Manipulates**
System Browser	Main design components
Relation Browser	Source data relations
Graphic Relation Browser	Graphic sources and graphic relations
Binding Browser	Interobject bindings
Output Binding Browser	Output bindings
Design Browser	Visualisation designs
Project Editor	Project operation
Select Editor	Select operation
Join Editor	Join operation
Equijoin Editor	Equijoin operation
Map Editor	Map operation
Sequence Editor	Sequence operation
Summarise Editor	Summarise operation
Tile Browser	Tiles and tile layers

Table 10 Tools in the Relational Visualisation Toolkit

http://www.cs.waikato.ac.nz/~matth/rvn. A prototype written in 'C' and using X graphics is also available at the site, along with support materials and examples.

5.1. USING THE TOOLKIT TO RECREATE MINARD'S GRAPHIC

To show that the Relational Visualisation Notation is accessible to users, the Relational Visualisation Toolkit is used to recreate Minard's Napoleonic visualisation. Minard's graphic portrays the movements of Napoleon's troops against the background of Russian geography. The data set underlying this visualisation consists of two relations: one for the movement of the troops and another for the temperature. The former describes the transit of soldiers from one city to another, whether they were advancing or retreating, when they departed, and how many perished along the way. The latter indicates the temperature on particular days. The data are graphically represented as a line segment of varying width connecting the cities of a geographic map of Russia. The temperature data appear as a line graph beneath the map, the horizontal axis showing location rather than time. Vertical lines connect the temperature points to the cities at which the measurements were taken.

These features are modelled by four components within the toolkit. The data are represented by two data relations *Napoleon* and *Temperature*. A fragment of Napoleon appears in Figure 22. The Russian geography and city positions will be represented by a graphic source called *Russia*. and described in Section 5.2. The top-level design diagram called *Minard* is prototyped in Section 5.3 by combining the source relations and the graphic source with relational operators. Section 5.4 shows several stages in the creation of the *movement* graphic relation that visualises the movements of troops. The final design, which include a temperature graph indexed the movement of troops, appears in Section 5.5. The generated visualisation has been shown previously, in Figure 21; the design is used with alternative data in order to visualise Hitler's 1941-1942 Central Russian campaign in Section 5.5.

5.2. CREATING A GRAPHIC SOURCE

To produce Minard's visualisation, the troop movement information must be augmented with a background map of Russia and the locations of the cities on that map. Rather than having to specify the city coordinates directly, as required by other toolkits [Roth 94], the cities may be directly identified as points on the map. A graphic source permits this because it can generate relational data from a background picture.

The initially empty graphic source *Russia* can be edited with the Graphic Relation Browser to provide a map of Russia populated with cities. At this stage the graphic source, shown in Figure 23, contains a text line with the title, a segmented line representing the Niemen River and a river title text line. These objects form the background image against which the visualisation will be constructed. The graphic source also contains a figure tuple (the dashed rectangle) that will be used to represent the cities. In addition to providing a backdrop, the graphic source produces relational output. The data have three attributes: the name of the city,

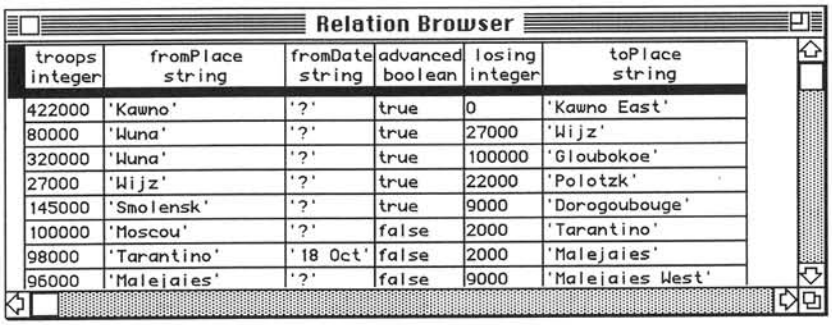

Figure 22 Relation Browser

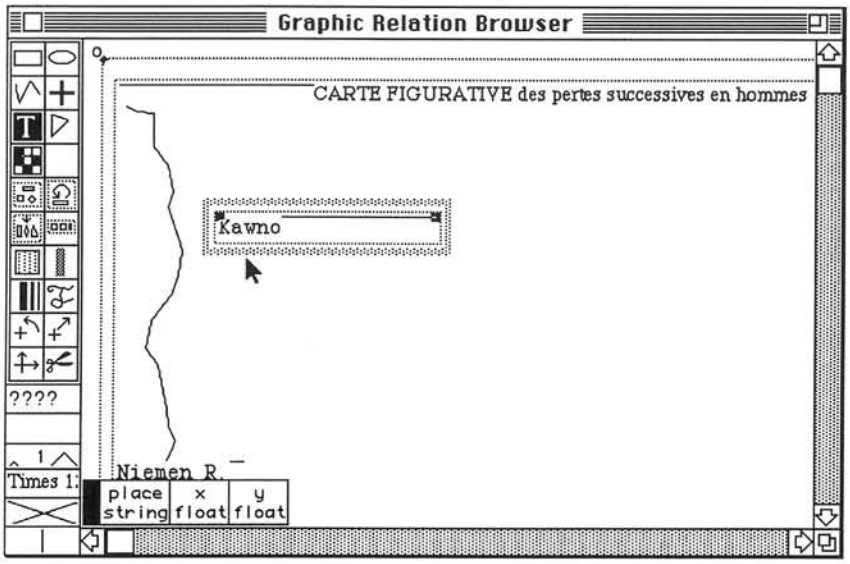

Figure 23 Editing text object in the figure tuple

its *x* location and *y* location.

The data of a graphic source are produced from figure tuples—groups of graphic objects that visually represent the source information. Bindings create the information by relating graphical attributes of the image to the output attributes. Each city is modelled by a figure tuple, and creating the cities begins by editing the first figure tuple. A text object is created in the figure tuple and the text contents are given as *Kawno*, one of the Russian cities. The text object is named *city*, which will be used later to create the bindings. The thin dotted grey line outlines its bounds—were there several objects, the bounds would encompass them all.

Next, three output bindings are created—one for each output attribute. The *place* attribute is bound to the expression *city.contents*. That is, the textual contents of the *city* object provide the value of the output attribute *place*. The *x* attribute is bound to the expression *city.amidx*—the horizontal midpoint of the city text name. The *y* attribute is similarly bound to *city.amidy*.

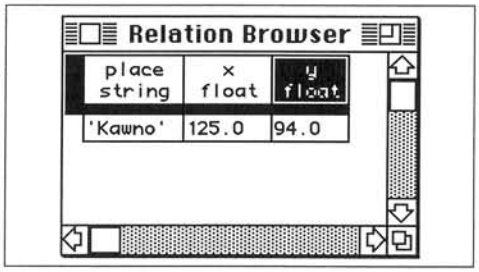

Figure 24 The generated relation

With the bindings in place, the graphic source's

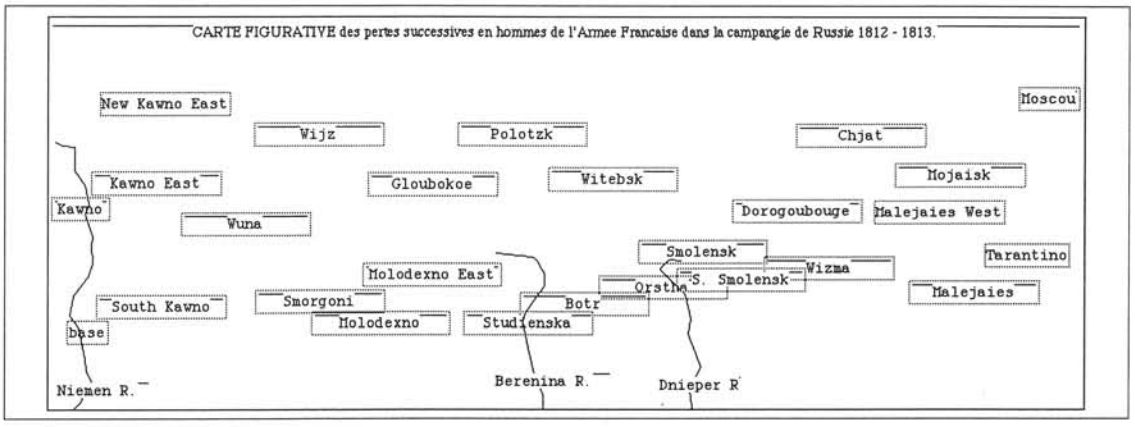

Figure 25 Final map of Russia with all figure tuples

relational output can be examined. Figure 24 shows the relation that is produced automatically. It has the same three attributes held by the graphic relation, and the sole tuple corresponds to the figure tuple in the picture. The coordinates of the city are derived from the position of the text on the screen. If the text object is moved to a new position, the output will automatically change to correspond.

Now that the first city has been prototyped, the remaining ones can be produced by duplicating it, dragging it to a new position and changing its the text value. Each copy has the same bindings as the original—no new bindings need to be specified. City names and positions can be adjusted as necessary without regard for their exact coordinates. Two more rivers and several cities later the entire graphic source is in place, shown in Figure 25. Each city produces one tuple and all of the cities appear in the relation. The final data generated from the image appear in Figure 26. Only decorative details and titles remain to be added.

Relation Browser

place string	x float	y float
'Kawno'	30.5	157.0
'Kawno East'	89.5	137.0
'New Kawno East'	96.5	75.0
'Wuna'	159.5	169.0
'Wijz'	216.5	99.0
'Gloubokoe'	305.5	138.0
'Orstha'	486.5	219.0
'Malejaies'	729.5	223.0
'Tarantino'	770.5	194.0
'Mojaisk'	718.5	131.0
'Polotzk'	375.5	99.0
'Witebsk'	446.5	134.0
'Moscou'	787.5	71.0
'Dorogoubouge'	591.5	159.0
'Smolensk'	517.5	191.0
'Chjat'	641.5	100.0
'base'	36.0	254.0
'Malejaies West'	702.5	160.0
'Wizma'	616.5	204.0
'Smorgoni'	217.5	230.0
'South Kawno'	93.5	234.0
'Botr'	424.5	232.0
'Studienska'	380.5	247.0
'Molodexno East'	305.0	209.0
'Molodexno'	265.0	247.0
'S. Smolensk'	547.5	213.0

Figure 26 Data from Russia

5.3. CREATING A DESIGN DIAGRAM

The Design Browser allows users to create, edit and test visualisation design graphs through incremental prototyping. A design does not need to be specified all at once. Rather, progressive stages successively achieve particular effects. Figure 27 shows the design browser with three sources and a join operator already added.

Joining the Russia and Napoleon relations computes the new data. The join associates the city position with each troop movement. A join operator is added to the design with arrows that connect its inputs to the source relations. It selects those tuple pairs for which the map's place-name matches the *fromPlace* of the troop movement, and thereby associates the coordinates for the place name with the matching place name troop movement. The Design Browser automatically computes the output schema of the join and displays it.

The selection criterion of the join is specified using the Join Editor, shown in Figure 28, which is activated by double-clicking on the node. Relational operators requiring parameters have their own specialised editors. The first line contains the schemas of the input relations. The second line shows the derived output schema, which is the concatenation of the input schemas. The text box contains the join condition which must be true of all output tuples. In

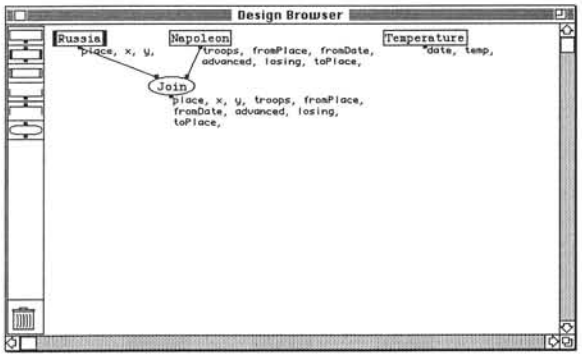

Figure 27 Join of data sources

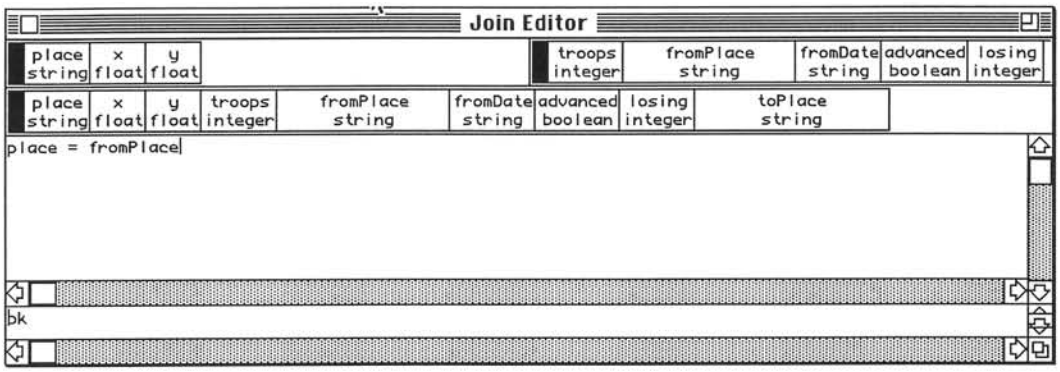

Figure 28 Editing the join conditional expression

this case, *place* (from the map) must match *fromPlace* (from the troop movements). The "ok" on the bottom line indicates that the criterion is syntactically correct.

A second join can be added in order to determine the coordinates of the destination cities. The selection condition of this join is *left_place* = *toPlace*, to match the names of the Russian cities on the map with those of the destinations of troop movements. The condition is entered again through the join editor. The output of the second join contains the city names, which are not needed in order to visualise the troop movement because they have already been translated to coordinates. The *left_place*, *right_place*, *fromPlace* and *toPlace* attributes can therefore be removed, along with the dates of travel by using a project operator.

After adding the project operator and setting its conditions, the output of the design can be checked for correctness. The Design Browser allows the relation associated with any output pier to be examined: Figure 29 shows the one produced from the *project* node. It contains the correct troop movement data and coordinates and can be used by a graphic relation to visualise the troop movements.

Relation Browser

right_x float	right_y float	left_x float	left_y float	troops integer	losing integer	advanced boolean
30.0	157.0	89.0	137.0	422000	0	true
89.0	137.0	106.0	79.0	22000	0	true
89.0	137.0	159.0	169.0	400000	0	true
159.0	169.0	221.0	94.0	80000	27000	true
159.0	169.0	305.0	138.0	320000	100000	true
547.0	213.0	486.0	219.0	37000	13000	false
770.0	194.0	729.0	223.0	98000	2000	false
787.0	71.0	770.0	194.0	100000	2000	false
641.0	100.0	718.0	131.0	127000	27000	true
221.0	94.0	375.0	99.0	27000	22000	true
305.0	138.0	446.0	134.0	220000	45000	true
718.0	131.0	787.0	71.0	100000	0	true
517.0	191.0	591.0	159.0	145000	9000	true
446.0	134.0	517.0	191.0	175000	30000	true
591.0	159.0	641.0	100.0	136000	9000	true
93.0	234.0	36.0	254.0	10000	0	false
729.0	223.0	702.0	160.0	96000	9000	false
702.0	160.0	616.0	204.0	87000	22000	false

Figure 29 Intermediate output of troop movement

5.4. CREATING A GRAPHIC RELATION

A graphic relation is needed to visualise the movements of troops from one city to another. It takes the coordinates of the origin and destination cities, the number of troops, the number of casualties and whether or not the troops were advancing, and uses this data to produce a polygon that stretches from one city to another. The ends of the polygon are perpendicular to the axis of movement. The widths of the ends are proportional to the number of troops that left and arrived. The polygon has no edging or frame, but is filled with a light shade if the troops are advancing and a dark one if they are retreating.

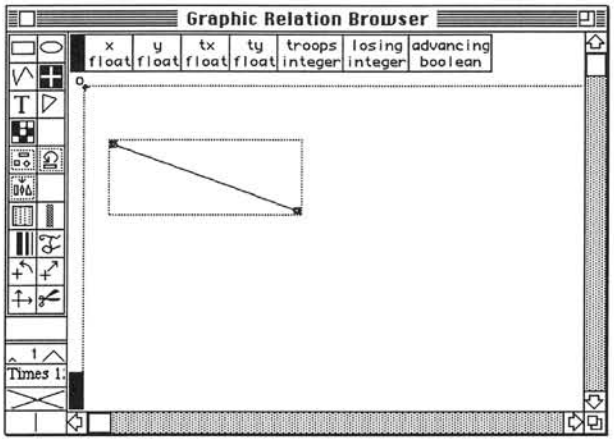

Figure 30 Axis of movement between points

A new graphic relation, called *movement*, is created with the Graphic Relation Browser and given the schema of the incoming data, as shown in Figure 30. The attribute names do not need to match those used in the design, but must have the same domains in the same order; matching is by position. The graphic relation is begun with a line segment that connotes the troops moving from one city to the next. It is named *axis* so that it can be referred to in bindings. It will not appear in the final result, because it merely forms a guideline to help create the final polygon.

The guideline axis is bound directly to the input coordinates. The *xfrom* and *yfrom* line segment attributes are bound to the *x* and *y* input attributes, which represent the departure city. The *xto* and *yto* object attributes are bound to the *tx* and *ty* input attributes, which represent the troop arrival city.

Figure 31 shows two more line segments for the ends of the polygon. The left is named

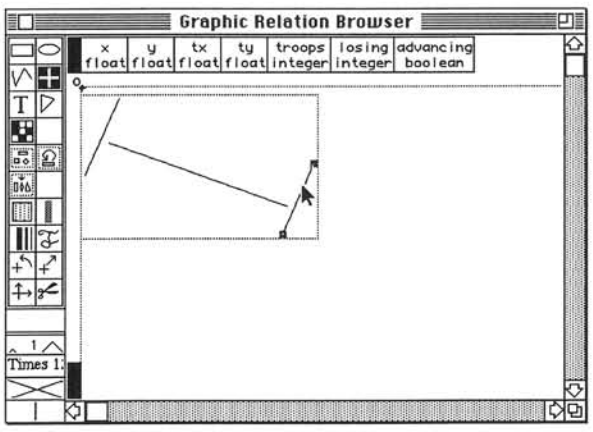

Figure 31 Adding perpendicular end-segments

```
fromLine.length := 75 * troops / 422000
fromLine.angle := axis.angle – pi / 2.0
fromLine.putfx := x – (75 * troops / 422000) * cos (axis.angle – pi / 2) / 2
fromLine.putfy := y – (75 * troops / 422000) * sin (axis.angle – pi / 2) / 2

toLine.length := 75 * (troops – losing) / 422000
toLine.angle := axis.angle – pi / 2.0
toLine.putfx := tx – (75 * (troops – losing) / 422000) * cos (axis.angle – pi / 2) / 2
toLine.putfy := ty – (75 * (troops – losing) / 422000) * sin (axis.angle – pi / 2) / 2
```

Figure 32 Troop movement line bindings

fromLine and the right is named *toLine*; they represent the troop populations that leave from one city and arrive at another. The new line segments must be perpendicular to the guideline axis, and their lengths must be proportional to the number of troops at the respective ends. A further constraint requires that the axis cross the line segments at their midpoints. The bindings of Figure 32 enforce these constraints.

The first two bindings set the length and angle of the line. The *length* binding uses an expression to translate the number of troops to the width of the line. The second two bindings assign the *putfx* and *putfy* attributes that move the line segment as a whole. They position the midpoint of the line segment at the end point of the axis. They convert the coordinates of the midpoint of the line *x, y* into the end-point of the line, using the line's length and angle. Similar bindings apply to *toLine*, although the troop population here discounts those that perished along the way.

Minard's troop movements are a continuous path of varying width, which we approximate with quadrangular polygons whose vertices correspond to the end points of *fromLine* and *toLine*. Figure 33 shows such a 4-point polygon. The polygon is stretched to fit the frame by binding its points to the end points of the line segments. A polygon binds all of its points in one step by loading them from a text string. Coordinates are separated by semicolons, the values themselves being separated by commas—for example, "34,12; 56,12; 78,12".

The binding expression for the polygon is:

fromLine.xfrom + "," + *fromLine.yfrom* + ";"+
fromLine.xto + "," + *fromLine.yto* + ";" +
toLine.xto + "," + *toLine.yto* + ";" +
toLine.xfrom + "," + *toLine.yfrom*

The binding *fill := if advancing then grey else verydarkgrey endif* sets the fill pattern so that advancing troops are coloured grey and retreating troops are very dark grey. The words *grey* and *verydarkgrey* refer to constants that are predefined texture bitmaps.

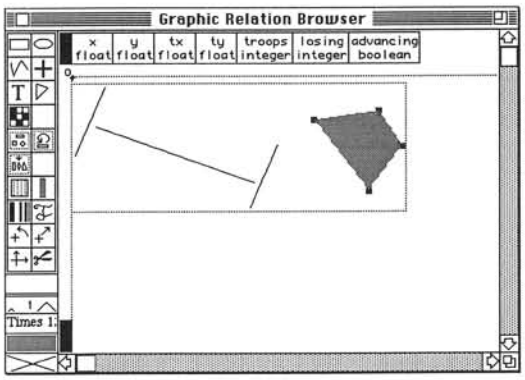

Figure 33 Adding a filled polygon to the axis frame

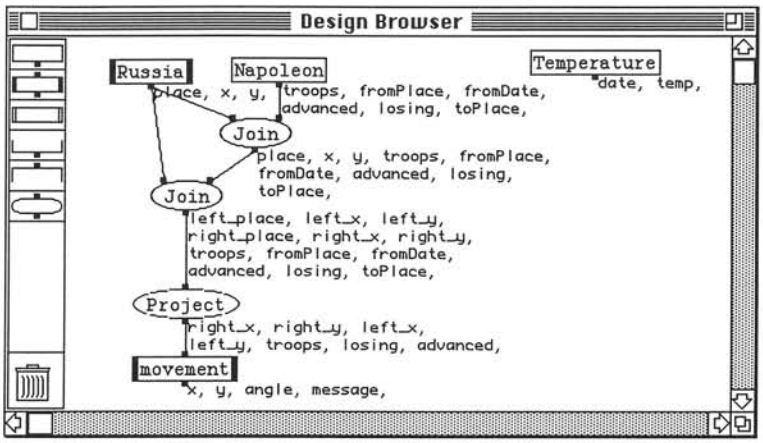

Figure 34 Design with movement graphic relation

The graphic relation can now be attached to the main design. Figure 34 shows the Design Browser with the movement graphic relation added and connected to the movement data. The output of the project attaches directly to the input of the graphic relation. The new design can be displayed with the actual data: the result appears in Figure 35. The cities are now connected by polygons whose thickness is proportional to the troop numbers.

5.5. COMPLETED VISUALISATION DESIGN

The last component of Minard's graphic is the temperature scale that shows the freezing weather that the soldiers endured. Visualising the temperature requires a two new graphic relations, a graphic source to represent the axes, and several new operators. The graphic source axis and graphic relation data points are woven into the visualisation design as shown in Figure 36. The temperature data is joined with the troop movement data where the coordinates of the departure cities are known—attaching the temperatures to the cities where the troops were at. The result is projected to essential line graph information: date, temperature and city location. This is paired with the vertical axis data and the result is then visualised as distinct data points.

The generated visualisation appears in Figure 21. The result closely replicates Minard's original. The troop path is very similar—it follows the same course with the same deflections at

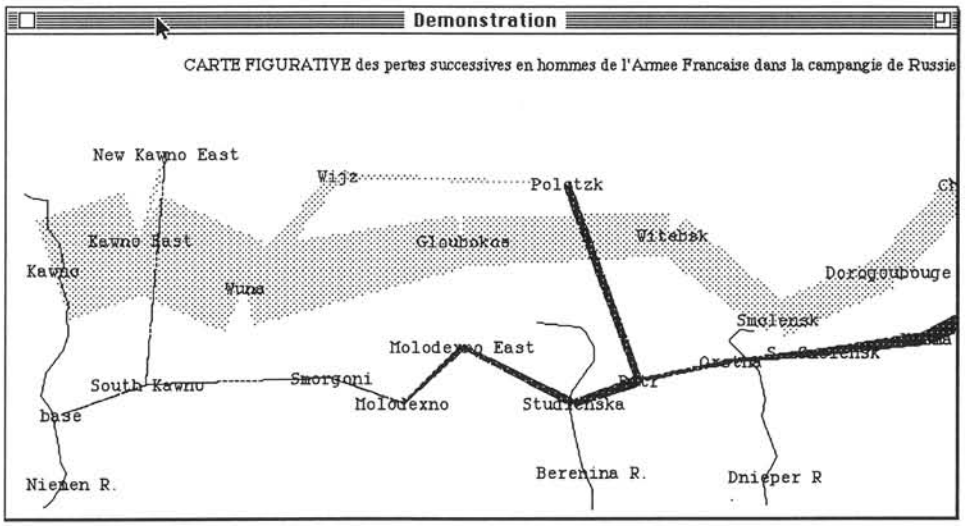

Figure 35 Trial output

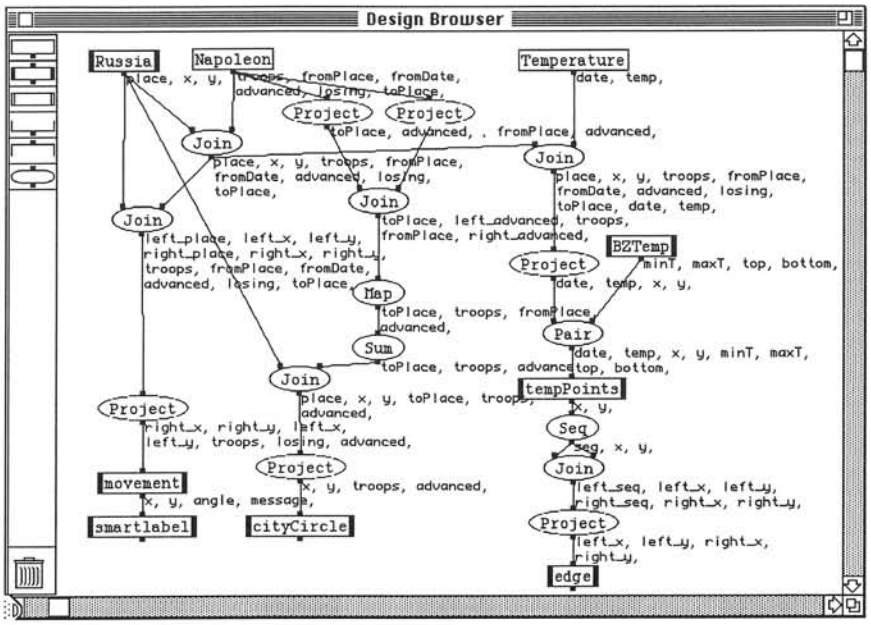

Figure 36 Complete design including temperature

cities. It has the same shading for advancing and retreating movements. The troop path branches similarly to the original, and rejoins it correctly. The troop path is smooth and continuous—polygons in other reconstructions appear disjointed [Roth 94].

The temperature graph mimics the original, including the axis labels, and the dates. The troop population labels are visible and have the same values as the original. The titles and geography are also the same, but these are specified directly by the user. The user has complete control over their appearance. Even the cities are placed directly by the user; there is no need to determine or specify their coordinates.

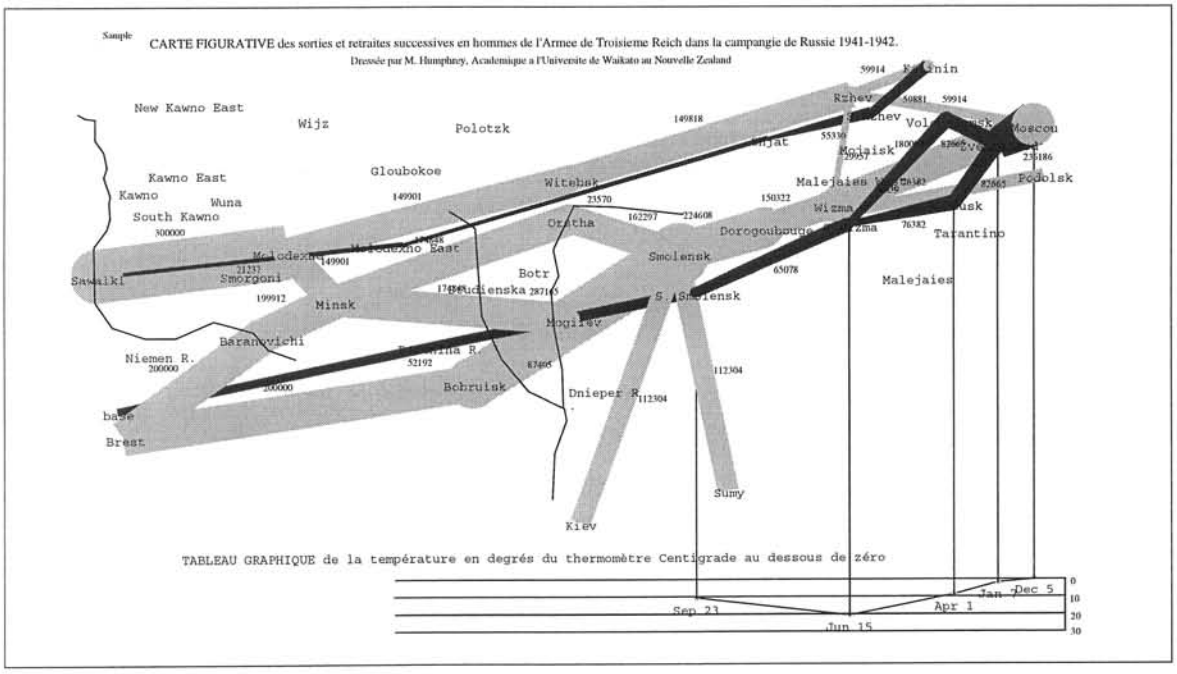

Figure 37 Hitler's 1941-1942 Central Russian Campaign

Figure 37 uses Minard's visualisation design with data from Hitler's 1941-1942 Central Russian campaign. The movement of troops is much more complicated here, with several outbranchings and rejoinders. Some troops, like those going to Kiev, left the central Russian theatre and did not return. Nevertheless, the new visualisation clearly shows the encirclements of Minsk and Smolensk and the rapid successes that the Wehrmacht enjoyed until it reached Moscow. After a long siege on Moscow and its environs, the German forces eventually withdrew, taking considerable losses.

The functionality of the Minard design was not modified in order to produce this diagram. The relational design is the same as for Napoleon's data, and all of the graphic relations are the same. In fact the visualisation is produced from the same specific design. Several circumstantial changes have been made however. New cities needed to be added to the map of Russia, to account for those appearing in the data. These were created by copying and renaming existing cities; no new bindings or functional units were created. The Russian geography was updated to correspond to contemporary 1942 maps and the titles were changed. In effect, only non-formal "decorative" information components were updated.

6. Conclusion

The Relational Visualisation Notation allows a very wide range of complex visualisation designs to be specified. Its expressiveness covers many known display styles and encourages the discovery of new designs. Unusual, novel or creative graphical representations that would have required specialised programs to produce can now be designed and executed with a single specification. The notation is also convenient, compact and easy to use; it does not rely on programming structures but achieves its flexibility through widely used relational concepts.

The notation is computationally powerful: it can represent complex transformations and visual relationships. Through relational modelling a very broad range of information structures and relationships can be represented. Relational algebra supports complex computations and data derivations. Furthermore, it gives the notation a formal and rigorous basis for reasoning about visualisations. Sequencing and summarisation operators provide rich computational facilities for deriving complex values. Replication and selection graphic groups provide flexibility for producing visualisations with varying components.

Moreover, the notation is accessible: it is supported by a suite of direct-manipulation, interactive design tools and usable by people without programming skills or technical knowledge. Relational algebra is widely used in modern database packages. The notation is declarative, rather than procedural, and graphical rather than textual. It does not contain algorithmic elements, variables or memory state, recursive elements or iterative elements. Users do not need training in or understanding of these concepts to be able to specify designs.

Nevertheless, the notation has three main limitations. It cannot compute the transitive closure of any operation, it cannot express algorithms, and it cannot construct arbitrary data structures. These limitations are serious because they restrict the kinds of visualisations that can be produced. The lack of a transitive closure operator precludes the visualisation of arbitrarily deep hierarchically and recursively structured information. The lack of algorithmic specification means that existing algorithmic visualisation techniques, such as triangulation, must be converted to functional form. Without arbitrary data structures, some informational structures cannot be represented and some graphical structures cannot be created. These limitations are intrinsic to the notation's foundation in relational algebra.

Despite these limitations, relational algebra is still a powerful paradigm for data visualisation because the ultimate goal is for users to be able to create their own applications, without algorithmic programming. Graphical representations, like those produced by this notation, are frequently the basis of direct-manipulation user interfaces. We have provided a means for

users to create semantically relevant visual images. Now it remains to provide a general means for interacting with those images.

References

BARTH, P. (1986). An Object–Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, **5**, 142–172.

BECKER, R.A. & CLEVELAND, W.S. (1987). Brushing Scatterplots. *Technometrics*, **29**, 102-117.

BECKER, R.A., CLEVELAND, W.S. & WILKS, A.R. (1987). Dynamic Graphics for Data Analysis. *Statistical Science*, **2**, 355–395.

BERTIN, J. (1983). *Semiology of Graphics*, translated by W.J. Berg, University of Wisconsin Press, Milwaukee, Wisconsin, USA.

CABRAL, B. & HUNTER, C. (1989). Visualization Tools at the Lawrence Livermore National Laboratory. *COMPUTER*, August 1989, 77–84.

CARROLL, THOMAS & MALHOTRA (1980). Presentation and Representation in Design Problem Solving. *British Journal of Psychology*, **71**, 143–155.

CHERNOFF, H. (1973). The Use of Faces to Represent Points in k–Dimensional Space Graphically. *Journal of the American Statistical Association*, **68**, 361-368.

DRAPER, S.W., WAITE, K.W. & GRAY, P.D. (1990). Alternative Bases for Comprehensibility and Competition for Expression in an Icon Generation Tool. *Human–Computer Interaction—INTERACT '90*, D. Diaper, D Gilmore, G Cockton, & B Shackel (eds), Elsevier Science (North Holland), pp. 473–477.

GRAY, P.D., WAITE, K.W. & DRAPER, S.W. (1990). Do–It–Yourself Iconic Displays: Reconfigurable Iconic Representations of Application Objects. *Human–Computer Interaction – INTERACT '90*, D. Diaper, D. Gilmore, G. Cockton, & B. Shackel (eds), Elsevier Science (North Holland), pp. 639–644.

HAWRYSZKIEWYCZ, I.T. (1988). *Inroduction to Systems Analysis and Design*, Prentice Hall, New York.

INGALLS, D., WALLACE, S., CHOW, Y., LUDOLPH, F. & DOYLE, K. (1988). Fabrik: A Visual Programming Environment. *Proceedings of the 1988 Conference on Object–Oriented Programming Languages and Systems*, published in *SIGPLAN Notices*, Vol. 23, No. 11.

KAMADA, T. & KAWAI, S. (1991). A General Framework for Visualising Abstract Objects and Relations. *ACM Transactions on Graphics*, **10**, 1–39.

KAZMAN, R. & CARRIÈRE, J (1996). Rapid Prototyping of Information Visualizations using VANISH. *Proceedings of InfoVis '96*, San Francisco, CA, October 1996.

KOSSLYN, S.M. (1985). Graphics and Human Information Processing. *Journal of the American Statistical Association*, **80**, 499–512.

LARKIN, J. & SIMON, H. (1987). Why a Diagram is (Sometimes) Worth 10,000 Words. *Cognitive Science*, **11**, 65-99.

MODELL, M. (1992). *Data Analysis, Data Modelling and Classification*. McGraw–Hill, Software Engineering Series.

MYERS, B.A. (1987). Gaining General Acceptance for UIMSs. Proceedings of the SIGGRAPH '87 Conference, published in Computer Graphics, Vol. 21, No. 2, 1987, pp. 130-134.

MYERS, B.A. (1989). User-Interface Tools: Introduction and Survey. *ACM Transactions on Information Systems*, **12**, 15-23.

POULOVASSILIS, A. (1992). The Implementation of FDL, a Functional Database Language. *The Computer Journal*, **35**, 119–128.

RHYNE, J.R., EHRICH, R., BENNETT, J., HEWETT, T., SIBERT, J.L. & BLESER, T.W. (1987). Tools and Methodology for User Interface Development. *Computer Graphics*, **21**, 78-87.

ROTH, S. (1994). Interactive Graphic Design using Automatic Presentation Knowledge. *Human–Factors in Computing Systems*, CHI '94, pp. 112–117.

SHAW, A. (1994). *The Wolfram Research Newsletter for Mathematica Users*, Wolfram Research.

SHNEIDERMAN, B. (1983). Direct Manipulation: A Step Beyond Programming Languages. *Computer*, **16**, 57–69.

TSE, T.H. & PONG, L. (1991) An Examination of Requirements Specificaion Languages. *The Computer Journal*, **34**, 143–152.

TUFTE, E.R. (1983). *The Visual Display of Quantitative Information*. Graphics Press.

TUFTE, E.R. (1991). *Envisioning Information*. Graphics Press.

ULLMAN, J.D. (1983). Principles of Database Systems. Computer Science Press.

UPSON, C., FAULHABER, T., KAMINS, D., LAIDLAW, D, SCHLEGEL, D., VROOM, J., GURWITZ, R. & VAN DAM, A. (1989). The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, **9**, 30–42.

WAITE, K.W. & DRAPER, S.W. (1991). User Input to Iconographer. *People and Computers VI, Proceedings of the HCI '91 Conference,* August 20–23, pp. 186–198.

ZLOOF, M. (1975). Query–by–Example. *AFIPS Conf. Proc.*, **44**, NCC, AFPIS Press, Montvale, N.J.