Chapter 4
A Framework for Visualization

4.1 Introduction

A database system is able to manage many different data arrangements using a declarative description of the data, i.e., the schema, and generic components for storing and manipulating the data. To manage visual information such as schemas, a DTSM must take a similar approach: use declarative description of schemas and their visualizations, and generic methods to manipulate them. These cannot be developed in an ad hoc manner; the design and implementation of a customizable and extensible schema manager must be based on a formal understanding of the interplay between schemas and their visual presentations. To achieve this goal, we develop a formalism of the visualization process. It consists of the following main elements: 1) a *data model* that captures graph-like information on a conceptual level, 2) a *visual model* that captures visualizations of such information, and 3) a mapping between data and visual models, referred to as a *visual metaphor*. We use the term *visual metaphor* (abbreviated as *metaphor*) because *metaphor* has been used by the interface community in a variety of ways, generally describing a transformation between abstract and visual information [BCCL91].[2] The abstract information of concern to us is the database schema, but this formalism is applicable to any similarly structured information; throughout this section the word "schema" could be equally well replaced by "instance of graph-like information". Examples of visual models and metaphors for visualizing schemas include directed graphs, E-R diagrams, and textual tables. Clearly, these metaphors have different characteristics, and would be useful in different circumstances. In general, there is no ideal metaphor, thus metaphor choice is important. Unfortunately, no general metaphor selection criteria exist. For the specific case of database schemas (and similarly structured information), our formalism is intended to provide a framework for flexible use, definition, and evaluation of visual metaphors. This formalism permits the declarative definition of models and metaphors, provides criteria for the identification of incorrect metaphors, and presents some guidelines for metaphor comparison. It supports mixed metaphors, i.e., the use of different visual metaphors for different parts of a single schema, establishing when and how metaphors may be mixed. Finally, the formalism allows the richness of most visual models to be used to capture information that is meaningful to the user but is beyond the database schema. A Desk-Top Schema Manager called OPOSSUM has been implemented based on this formalism, and is described in detail in the next chapter. While our work is oriented towards database schemas, the formalism is applicable to the visualization of any structured data that conforms to our definition of data models (which is found in the next section).

4.2 Data and Visual Models

For completeness, before presenting the definition of data and visual models, we briefly review some basic definitions and notations of binary relations which will be used later in the definition of visual metaphors. A binary relation r from set A (its *domain*) to set B (its *range*) is a subset of $A \times B$ denoted $r : A \rightarrow B$. The relation r is called a *function* if for each $a \in A$ there is at most one $b \in B$ where $(a,b) \in r$; it is called *total* if for each $a \in A$ there is at least one $b \in B$ where $(a,b) \in r$; it is called *injective* if its inverse is a

[2] The term metaphor is sometimes used to describe behavior of visualizations as well as appearance. We do not consider behavior in this chapter.

function; and it is called *onto* if its inverse is total. Injective functions are sometimes called 1-1 functions. Binary relations can be unioned, in which case, their domains, their ranges, and the sets of pairs in the relations are unioned, respectively. For a function r, we write $r(a) = b$ instead of $(a,b) \in r$. Functions may be applied to sets of values; if $A' \subseteq A$, then $r(A')$ is equal to $\{r(a) \mid a \in A'\}$. We also use r^{-1} to denote the inverse of a function r, which may return a subset of A if r is not 1-1. Finally, for a function r, the notation $r(a)$ is valid even if there is no $b \in B$ where $r(a) = b$. In that case, when applying a set-valued function on $r(a)$, the empty set is returned.

4.2.1 Problem Formulation

By definition, a *schema* describes the conceptual structure of some information in a database, specified using the concepts of some *data model* (for clarity, such a schema will henceforth be referred to as a *data schema*). We are interested in the process of creating a visualization of a data schema. For this, we introduce the notion of a *visual model*. Like a data model, it describes the structure of some information, though its concepts are visual. As with data schemas, any visual representation that conforms to a visual model will be called a *visual schema*. Visual schemas can be used in two general ways: as a read-only visual representation of the data schema, or as an editable visualization that allows creation or manipulation of a data schema through changes to it.

Using the above notions, the problem of visual representation of data schemas may be stated formally as follows. Given a data model D, let $S(D)$ denote the set of valid data schemas that can be constructed based on that model. Similarly, let $S(G)$ denote the set of visual schemas that can be constructed based on a visual model G. The sets $S(D)$ and $S(G)$ are defined to be the *information capacities* [Hul86, MIR93] of the data model D and the visual model G, respectively. In order to create visual schemas that represent data schemas, we require a binary relation between $S(D)$ and $S(G)$, whose specific properties depend on the intended use of the visual schemas. Specifically,

1. If a visual schema of G is used only to view any data schema of D in its entirety, then an onto function must exist of the form $f : S(G) \rightarrow S(D)$, so that every data schema can be represented visually.
2. If, in addition, a visual schema of G is also used to update a data schema of D, then a total onto function must exist of the form $f : S(G) \rightarrow S(D)$, so that every visual schema can be uniquely interpreted as a data schema.

Clearly, not all such functions f that satisfy these properties are useful. Many are arbitrary mappings, with no obvious correspondence between the data schema and the visual schema. Our goal is to establish a relationship between the members of $S(D)$ and $S(G)$ so that when users view a visual schema, they can infer the data schema to which it maps. Thus, f should be derived from a correspondence between the features of the data and visual models, which would enforce a structural similarity between data schemas and visual schemas. This correspondence is a *visual metaphor* and is formally introduced in Section 4.3. A formal description of the features of data and visual models is given in the next subsection.

This problem formulation was motivated by the very similar problem of mapping schemas and data between different data models in a heterogeneous database system [MIR93]. The specific similarities and differences between the two problems are beyond the scope of this thesis.

4.2.2 A Formalism for Data/Visual Models and Schemas

In this subsection, we present a meta-model that can capture a large and interesting class of data and visual models. Using this meta-model, we can describe the features of any model in this class and discuss the relative information capacity of any pair of models. Example models described in this meta-model are given in the following subsection.

Definition 1 Every data or visual model $\mathcal{M}$ can be seen as a sextuple $\mathcal{M} = <\mathcal{P}, \mathcal{A}, \mathcal{V}, \mathcal{Q}, \mathcal{R}, \mathcal{C}>$ defined as follows:

$\mathcal{P}$ is a finite set of identifiers for the concepts in $\mathcal{M}$. Each such concept P is associated with a (possibly infinite) set of globally unique ids $\mathcal{I}(P)$ that can be used to identify concept instances of type P.[3]

$\mathcal{A}$ is a finite set of identifiers for property names (i.e., attributes) of concepts in $\mathcal{M}$. Each element of $\mathcal{A}$ is of the form $P.A$, where $P \in \mathcal{P}$ and A captures some attribute that all concept instances of type P should have. We use $P.A$ instead of just A because the same attribute name may be used by more than one concept.

$\mathcal{V}$ is a (possibly infinite) set of identifiers for values of properties of concepts in $\mathcal{M}$. Each element of $\mathcal{V}$ is a value binding of the form $P.A == v$, where $P.A \in \mathcal{A}$ and v captures some value that the $P.A$ attribute may have. The values v may be drawn from sets of basic values (such as integers or character strings), or the sets of concept instances.

$\mathcal{Q}$ is a function $\mathcal{Q} : \mathcal{P} \rightarrow 2^{\mathcal{A}}$, indicating for each concept $P \in \mathcal{P}$ the set of attribute identifiers in $\mathcal{A}$ that correspond to P.

$\mathcal{R}$ is a function $\mathcal{R} : \mathcal{A} \rightarrow 2^{\mathcal{V}}$, indicating for each attribute $P.A \in \mathcal{A}$ the set of value bindings in $\mathcal{V}$ that can be assigned to it. To capture the set of actual values instead of the value bindings (e.g., v instead of $P.A == v$) the function $\mathcal{R}^*$ is used, where $\mathcal{R}^*(P.A) = \{v \mid P.A == v \in \mathcal{R}(P.A)\}$

$\mathcal{C}$ is a finite set of constraints, i.e., rules that must be satisfied by any schema expressed in $\mathcal{M}$. These constraints are formulas in some prespecified language $\mathcal{L}$ and use elements that refer to identifiers in $\mathcal{P}$, $\mathcal{A}$, $\mathcal{V}$.

Note that, by the way $\mathcal{A}$ and $\mathcal{V}$ were defined, if $P \neq P'$ then $\mathcal{Q}(P)$ and $\mathcal{Q}(P')$ are disjoint, and if $P.A \neq P'.A'$ then $\mathcal{R}(P.A)$ and $\mathcal{R}(P'.A')$ are disjoint. Also note that concept instances are essentially complex objects, since some of their attributes can take values that are concept instances themselves.

Most of the common data models fall naturally in this meta-model. For example, consider the relational model. It has *relations* and *attributes* as its concepts, each *relation* has a *name*, and each *attribute* has a *name*, a *type*, and a *relation* with which it is associated. Fully specified examples of data and visual models may be found in Section 4.2.4. The meta-model may be enhanced with several additional characteristics of models in a straightforward way, e.g., with an identifier for the name of each instance of the model, but we avoid that for simplicity of presentation.

As defined above, a data schema or visual schema may be considered as an instantiation of a data or visual model, respectively. This is formally defined as follows:

---

[3]The symbols P and $\mathcal{P}$ are used to denote concepts and sets of concepts because in an earlier presentation of this work, concepts were referred to as “types of primitives.”

Definition 2 A schema S of a model $\mathcal{M}$ is defined as follows:

- For every $P \in \mathcal{P}$, there is a finite set $[P] \subseteq \mathcal{I}(P)$ of concept instances of type P that appear in schema S.
- For every $P.A \in \mathcal{A}$, there is a total function $[P.A] : [P] \rightarrow [\mathcal{R}^*(P.A)]$, which determines the value of the $P.A$ attribute for every concept instance in $[P]$. The $[\mathcal{R}^*(P.A)]$ set is defined such that, if $\mathcal{R}^*(P.A) = \mathcal{I}(P')$, for some $P' \in \mathcal{P}$, then $[\mathcal{R}^*(P.A)] = [P]$. Otherwise, $[\mathcal{R}^*(P.A)] = \mathcal{R}^*(P.A)$.
- For every $c \in \mathcal{C}$, there is a constraint $[c]$, constructed from c by replacing every $P \in \mathcal{P}$ by $[P]$, every $P.A \in \mathcal{A}$ by $[P.A]$, and every $P.A == v$ by v. All these constraints are satisfied by the schema.

In the following table, we summarize the notations introduced in Definitions 1 and 2:

Notation	Explanation
$\mathcal{P}$	The set of concepts
P	A concept
$\mathcal{I}(P)$	The set of identifiers for concept instances of type P
$[P]$	The set of concept instances of type P in a schema
$\mathcal{A}$	The set of attributes for all concepts in $\mathcal{P}$
$P.A$	The A attribute of concept instances of type P
$[P.A](p)$	The value of the $P.A$ attribute of the concept instance p
$\mathcal{V}$	The set of values for all attributes in $\mathcal{A}$
$P.A == v$	The element v as a value of the attribute $P.A$
$\mathcal{Q}(P)$	The set of attribute identifiers of concept instances of type P
$\mathcal{R}(P.A)$	The set of value identifiers of the $P.A$ attribute
$\mathcal{R}^*(P.A)$	The set of values of the $P.A$ attribute
$\mathcal{C}$	The set of constraints of a model
$[c]$	The instantiation of a constraint $c \in \mathcal{C}$ for the concept instances in $\mathcal{P}$ and attribute values $P.A$ in a schema

Table 4.1. Model Notation

4.2.3 Creating Visual Models

Creation of data models is a classical database problem that is beyond the scope of this chapter [BCN92]. In this subsection, we concern ourselves with creating suitable visual models. There is an important difference between the two kinds of models. Data models capture abstract organization of information. Their concepts, attributes, and values are determined by the information the model captures. Visual models, however, must reflect not only the information to be organized, but also the medium in which the models are expressed. Specifically, visual model concepts reflect both the information to be shown and the medium, while the possible attributes and values of a concept are determined by the medium alone. For example, consider a visual model used to display directed graphs. Any such model would likely have concepts corresponding to *nodes* and *edges*. If the model were oriented toward a monochrome ASCII terminal, these concepts and their attributes

and values would be very different from a similar model intended for a color bit-mapped display system. The ability to use colors, shapes, lines, and patterns would vary widely between the two. In general, the number and semantics of visual model concepts are determined by the information that must be displayed, but the precise composition of the concepts is determined by the medium.

Because data models are used to represent abstract information, their concepts may be chosen arbitrarily based on some conceptualization of the world. On the other hand, visual model concepts must be visualizable. Therefore, visual concepts must be constructed using only certain visual building blocks. Motivated by our involvement in developing a scientific Experiment Management System, we are concerned with visual models to be displayed on color bit-mapped workstations as these are commonly available to scientists. To build visual models for this medium, we have chosen the basic visual constructs described in the following table. The choice of these constructs is somewhat arbitrary. They are not formally defined in this chapter, but the interested reader may find a formal discussion of visual constructs elsewhere [FvDFH90]. In the table below, location is a complex attribute consisting of several coordinate values. For the region, text-display, and picture-display constructs, it is assumed to be the location of their center.

Construct	Attributes
region	shape, orientation, center-location, lower-left-location, background-color, background-pattern, boundary-width, boundary-color, boundary-pattern
line	source-location, dest-location, width, color, pattern
text-display	text, font, location, orientation, size, and color
picture-display	picture, location, orientation, size, and color

Table 4.2. Visual Constructs

Visual concepts are defined as *compositions* of the above constructs or other, previously defined, visual concepts. The attributes of a visual concept are the attributes of all of its components, possibly renamed to avoid any naming conflicts. Since compositions often have a large number of attributes, in our examples we have omitted many visual attributes to make the examples more manageable.

To make the appearance of a composition coherent, it will usually be necessary to include constraints relating the attributes of its different components. For example, if a box with a piece of text in the center were required, a composition of a *region* and a *text-display* would be defined as a concept, and a constraint would require the value of the location attribute of *text-display* to be the same as the value of the location attribute of *region*. These constraints regulate the appearance of visual concepts, and are called *composition constraints* to distinguish them from other, more semantically focused constraints.

4.2.4 Example Data and Visual Models

Consider a very simple semantic data model, supporting *entity-classes* that may be mutually related with binary *relationships*. Each entity-class has a *name* and a *kind*. The two possible kinds are ‘simple’ (such as the class of integers or the class of character strings) or ‘compound’ (user-defined classes). Each relationship has a *name*, a *card-ratio* of ‘1:1’, ‘1:N’, ‘M:1’, or ‘M:N’, and two entity-classes with which it is associated. This data model is the sextuple $D = < P_D, A_D, V_D, Q_D, R_D, C_D >$, where $C_D = \emptyset$, and the concepts in P_D, their attributes in A_D, and their corresponding value sets as determined by R^*_D are given in table 4.3.[4]

[4] For simplicity, we use the names of attributes directly instead of their corresponding full identifiers, i.e.,

Concept (P)	Attribute ($P.A$)	Attribute Values ($R^*(P.A)$)
entity-class	name	text
	kind	{simple, compound}
relationship	name	text
	card-ratio	{1:1, 1:N, M:1, M:N}
	from-class	I(entity-class)
	to-class	I(entity-class)

Table 4.3. An Example Data Model

Note that the set of values of an attribute has several possibilities: an infinite predefined set (e.g., text), an enumerated set (e.g., {1:1, …, M:N}), or the set of all instances of a concept (e.g., I(entity-class)).

Similarly, consider a very simple visual model that supports directed graphs. We define the concepts to be *nodes* and *edges*, the former a combination of a *region* and a *text-display*, and the latter a combination of a *line*, a *text-display*, and two *nodes*. This visual model is the sextuple $G = \langle P_G, V_G, A_G, Q_G, R_G, C_G \rangle$, where the concepts in P_G, their attributes in A_G, and their corresponding value sets as determined by R^*_G are given in table 4.4. For simplicity, only a subset of the attributes is shown. Note that some of the attributes whose allowed values are a set of size one. For example, the *label-color* attribute is only allowed to have the value "black." In these cases, the model is specifying that the attribute has a constant value.

Concept (P)	Attribute ($P.A$)	Attribute Values ($R^*(P.A)$)
node	shape	{square, oval}
	location	plane-points
	boundary-width	{2 pixels}
	color	{blue, red}
	label-text	text
	label-color	{black}
edge	source-location	plane-points
	dest-location	plane-points
	color	{black, blue, yellow, green, orange}
	from-node	I(node)
	to-node	I(node)
	label-text	text
	label-color	{black}

Table 4.4. An Example Visual Model

There are four constraints in set C_G that all schemas of G must satisfy. Two of them are composition constraints, related to the relative positioning of the region and text-display for *nodes*, and the line and text-

A instead of *P.A*. This also holds for all other models presented in this chapter and applies to any constraints that are shown as well.

display for *edges*. The remaining two are somewhat more interesting, determining the location of *edge* concept instances in terms of the *nodes* they connect. Using simple Horn-clauses, we show these constraints, which indicate that the location of the source (resp. destination) of an edge is the same as the location of the from-node (resp. to-node) of the edge:

$\forall e \in$ edge, source-location(e) = location(from-node(e)), and
$\forall e \in$ edge, dest-location(e) = location(to-node(e)).

4.3 Visual Metaphors

4.3.1 Definitions and Notation

A visual metaphor is defined as a correspondence between some of the features of a data and a visual model, i.e., elements in P, A, V. A metaphor induces a mapping between data schemas (instances of the data model) and visual schemas (instances of the visual model). Basing the schema mapping on the feature correspondence helps produce visual schemas that, when viewed, allow the user to deduce the underlying data schema (Section 4.2.1). Consider a data model $D = < P_D, A_D, V_D, Q_D, R_D, C_D >$ and a visual model $G = < P_G, A_G, V_G, Q_G, R_G, C_G >$. A metaphor will include correspondences between concepts (P_D and P_G), between attributes (A_D and A_G), and between attribute values (V_D and V_G). (The above define a correspondence between constraints as well, since constraints refer to elements of the P, A, and V sets.) These correspondences describe the meaning of visual model features with respect to the underlying data model. For example, given the data and visual models from Section 4.2.4, if a correspondence were defined between the concepts *entity-class* and *node*, then every instance of a *node* in a visual schema would imply the existence of a *entity-class* in the data schema. To allow presentation flexibility, we permit correspondences to exist between multiple features in the visual model and a single feature in the data model (an example of this is given in the next section). This is possible only when the visual model has a greater information capacity than the data model (Section 4.2.1), which is almost always the case.

Definition 3 A metaphor T is an onto function from G to D (denoted by $T : G \rightarrow D$), which is the union of the following three onto functions:

Function $T_p : P_G \rightarrow P_D$.

Function $T_a : A_G \rightarrow A_D$, which is equal to $\bigcup_{P \in P_G} T_a^P$, where for each concept P, $T_a^P : Q_G(P) \rightarrow Q_D(T_p(P))$ is an onto function.[5]

Function $T_v : V_G \rightarrow V_D$, which is equal to $\bigcup_{P.A \in A_G} T_v^{P.A}$, where for each attribute $P.A$, $T_v^{P.A} : R_G(P.A) \rightarrow R_D(T_a(P.A))$ is an onto function.

As mentioned above, all constraints in C_G use elements that refer to identifiers in P_G, A_G, and V_G. We occasionally use the notation $T(c)$, $c \in C_G$, for the constraint constructed from c by replacing each element

[5]Note that if P does not have an image under T_p, then $Q_D(T_p(P))$ is the empty set. Therefore T_a^P is empty as well which makes it vacuously an onto function. Similar observations hold for $T_v^{P.A}$.

referring to an identifier x of $P_G \cup A_G \cup G_G$ by an element referring to the identifier $T(x)$. We also use the notation $T(I_G(P))$ to denote $I_D(T(P))$.

4.3.2 The Induced Schema Mapping

Given a metaphor as defined above, a mapping between data and visual schemas can be induced. Using this induced mapping, any data schema of D can be transformed to a visual schema of G in a manner that remains faithful to the metaphor.

Definition 4 Given a metaphor $T : G \to D$ and a visual schema of G, T induces an onto function t from $\left(\bigcup_{P \in P_G}[P]\right) \cup \{[P.A] \mid P.A \in A_G\}$ onto the corresponding features of some data schema of D with the following characteristics:

- $(\forall P \in P_G)(\forall p \in [P])$ if $T_p(P)$ is defined, then $t(p)$ is a concept instance of type $T_p(P)$.
- $(\forall P \in P_G)(\forall P.A \in Q(P))$ if $T_a(P.A)$ is defined, then $t([P.A])$ is a function $[T_a(P.A)] : T_p(P) \to [R^*_D(T_a(P.A))]$ such that $(\forall p \in [P])$ the following holds:
  if $[P.A](p) = v$ and $T(P.A == v) = (P'.A' == v')$
  then $t([P.A])(t(p)) = v'$.

The first clause above states that concept instances of the visual schema in G represent concept instances in some data schema in D based on the type correspondence specified by T. The second clause states that, for every visual model attribute mapped by T_a, there exists a data model attribute whose values are determined based on the value correspondence T_v. Essentially, this is a commutivity requirement that is best shown in Figure 4.1. Based on the two clauses above, the induced function t determines a mapping from any visual schema in G to some data schema in D. As with the metaphor T, the induced function t can be extended to include in its domain instantiations of constraints, $t([c])$ for $c \in C_G$.

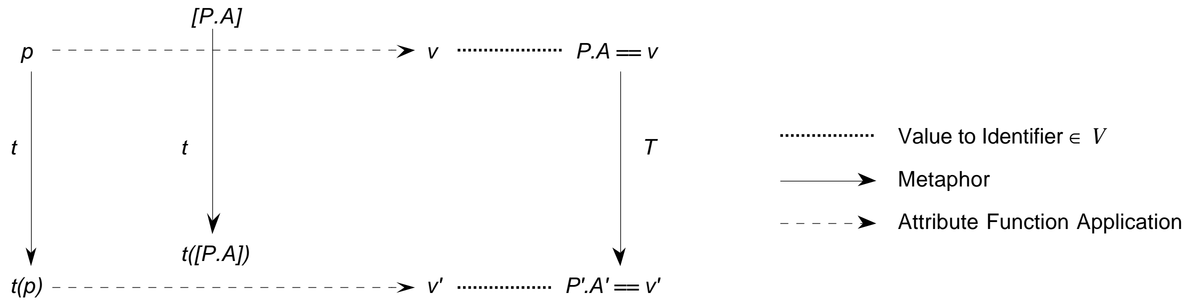

Figure 4.1. Commuting diagram between metaphors and attribute functions.

4.3.3 An Example

To illustrate the above definitions, we present a metaphor T. This metaphor maps from a visual model similar to that in Section 4.2.4 to the data model discussed in the same section. The visual model has been supplemented with an additional concept, called a blob, which consists of a region and two text-displays, referred to as label1 and label2. Also, the from-node and to-node attributes of relationship have been extended to accept as values both nodes and blobs. This function T is defined in the Table 4.5.

x	$T(x)$
node	entity-class
node.label-text	entity-class.name
node.label-text==x	entity-class.name==x
node.color	entity-class.kind
node.shape	entity-class.kind
node.color==‘blue’	entity-class.kind==‘simple’
node.shape==‘square’	entity-class.kind==‘simple’
node.color==‘red’	entity-class.kind==‘compound’
node.shape==‘oval’	entity-class.kind==‘compound’
blob	entity-class
blob.label1-text	entity-class.name
blob.label1-text==x	entity-class.name==x
blob.label2-text	entity-class.kind
blob.label2-text==‘P’	entity-class.kind==‘simple’
blob.label2-text==‘C’	entity-class.kind==‘compound’
edge	relationship
edge.label-text	relationship.name
edge.label-text==x	relationship.name==x
edge.from-node	relationship.from-class
edge.from-node==c	relationship.from-class==t(c)
edge.to-node	relationship.to-class
edge.to-node==c	relationship.to-class==t(c)
edge.color	relationship.card-ratio
edge.color==‘green’	relationship.card-ratio==‘1:1’
edge.color==‘orange’	relationship.card-ratio==‘1:1’
edge.color==‘yellow’	relationship.card-ratio==‘1:N’
edge.color==‘blue’	relationship.card-ratio==‘M:1’
edge.color==‘black’	relationship.card-ratio==‘M:N’

Table 4.5. A Sample Metaphor.

4.3.4 Discussion

A metaphor provides meaning to features of a visual model by establishing a correspondence between them and the features of a data model. The precise meaning is captured by the function T. For example, displaying a red oval *node* implies the existence of a compound *entity-class* in the data model. The use of the T function and its induced schema mapping t should produce visual schemas that users can correctly and unambiguously interpret. Depending on various properties of T, the visual schema may include features that do not carry any meaning and/or features that carry redundant meaning. Based on knowledge of T, users should be able to ignore the former and not be confused by the latter.

We would like to comment on the various properties of metaphors as they relate to the relative information capacity of a data and a visual model. As a minimum requirement, a metaphor has been defined as an onto function: if it were not onto, then some characteristics of a data model would not be captured visually; if it were not a function, then a single visual construct could have multiple meanings, and therefore could not be

interpreted correctly.[6]

For a metaphor that is not total, some visual elements do not mean anything with respect to the data model. If a metaphor will be used only for retrieving a data schema, T does not have to be total. Under retrieval, only existing schemas will be visualized, so non-totality does not create any problem. Specifically, non-totality of T_p or T_v implies that some visual concepts or some values of visual attributes respectively will never be used, while non-totality of T_a implies that the values of some attributes can be arbitrary. If a metaphor will be used for retrieving and updating a schema, T_p and T_a still do not have to be total. For T_a, the reasons are the same as above. A non-total T_p is permissible because visual concepts that are not mapped by T_p may be used for presentation purposes and can be ignored when mapping the visual schema to a data schema. For each $P.A$ that is mapped by T_a, however, $T_v^{P.A}$ must be total. Otherwise, one could draw a visual schema that would not be translatable to a data schema.

To demonstrate the use of a non-total metaphor, consider a data model D capturing words in the English language as strings of letters from the Roman alphabet. A visual model G is constructed to visually present these words based on a straightforward metaphor that maps each display of a word to the word itself. Because the strings are visually expressed, G must also include information about typeface, size, color, letter spacing, and other visual characteristics of letters that carry no particular meaning, i.e., T_a is not total. Hence, G has a greater information capacity than D: the number of its visual schemas is equal to the number of English words (about 600,000) multiplied many times by the possible typefaces, sizes, and the other characteristics. Yet no matter how font or size vary (within reason), a visualization of a word carries an unambiguous meaning in the context of the metaphor.

If a metaphor is not 1-1 then multiple visual elements have the same meaning with respect to the data model. For both retrieval and update, the implications of this are the same. If T_p or T_v are not 1-1 then there is a choice of visual constructs that can be used, which should be left to the user or resolved via some default mechanism. If T_a is not 1-1 then there is redundancy: multiple visual attributes capturing the same data attribute. By the nature of visual models, there is no issue of choice here: all attributes of a concept instance must have some value in a visual schema, and therefore all those mapped to the same data attribute should be assigned *consistent* values based on T_v. This issue of consistency arises because of the redundancy semantics. For visual updates of schemas, if several visual attributes are mapped to the same attribute by T_a, as soon as the value of one of them is specified, the values of all others are uniquely determined.

Functions that are not 1-1 establish equivalence classes among the features of the visual model, i.e., several features have the same meaning. For example, in the metaphor of Section 4.3.3,

$$T_p(\text{node}) = T_p(\text{blob}) = \text{entity–class}$$

implies that a concept instance of type *entity-class* may be represented equivalently as either a visual concept instance of type *node* or one of type *blob*. Attribute correspondences are similar but more complex. A data model attribute may correspond to a set of visual attributes, some of which may come from the same concept. For example, in the same metaphor,

$$T_a(\text{node .color}) = T_a(\text{node.shape}) = T_a(\text{blob.label2–text}) = \text{entity–class.kind}$$

indicates the same choice of *node* or *blob* concepts as above. In addition, it specifies redundancy: visual

[6] Non-onto functions and non-functional correspondences will prove useful in extending metaphors to allow visualizations of subsets of data schema information, part of our future work.

attributes *node.color* and *node.shape* are from the same concept, so they redundantly capture data attribute *entity-class.kind*. Value mappings are similar, but even more complicated, so they warrant two examples. First,

$$
\begin{aligned}
T_v(\text{edge.color} == \text{'green'}) &= T_v(\text{edge.color} == \text{'orange'}) \\
&= (\text{relationship.card–ratio} == \text{'1:1'})
\end{aligned}
$$

demonstrates attribute value choice; two values have the same meaning with respect to the metaphor. Second,

$$
\begin{aligned}
T_v(\text{node.color} == \text{'blue'}) &= T_v(\text{node.shape} == \text{'square'}) \\
&= T_v(\text{blob.label2–text} == \text{'P'}) \\
&= (\text{entity–class.kind} == \text{'primitive'})
\end{aligned}
$$

includes values from different concepts (e.g. *node.color* versus *blob.label2-text*), paralleling the choice at the concept level, and different values for different attributes of the same concept (e.g. *node.color* and *node.shape*), indicating values for redundant attributes.

Figure 4.2 gives an example schema and a visual schema that could be produced by applying the induced mapping of the example metaphor in Section 4.3.3. (Due to limitations of the printing medium, color attributes cannot be displayed directly; instead they are indicated by the name of the color along the line of the edge.)

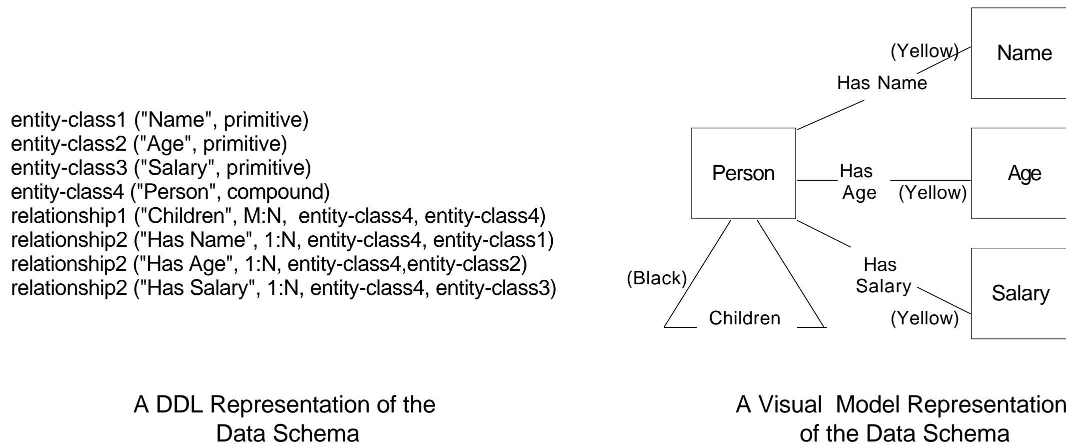

Figure 4.2. Example of a metaphor applied to a schema.

4.4 Judging Metaphors: The Good, the Bad, and the Ugly

Given a framework for creating visual metaphors, it is necessary to examine issues of metaphor correctness. We thus develop criteria that are useful in ensuring that a metaphor accurately presents information. There are issues beyond correctness, however, that affect how well metaphors visualize information. We discuss these issues of metaphor *quality* and their impact on visualization as well.

4.4.1 Metaphor Correctness

We have already discussed the requirements for the relation T in order for it to be a valid metaphor based

on the desired operational goals, retrieval and/or update. In addition, there are three other issues that affect the correctness of a metaphor. All three require some consistency between the metaphor and the visual model, the first in terms of allowed attribute values, and the second two in terms of constraints.

First, consider attributes whose values are concept instances, e.g., the *from-class* attribute of *relationship* in Section 4.3.3. It is necessary that the type of such a concept instance-valued visual attribute be consistent with the metaphor and with the type of the data model attribute to which it is mapped. Continuing the above example, the attribute *relationship.from-class* takes values of type *entity-class*. The concept *entity-class* is mapped to by more than one visual concept, specifically *node* and *blob*. As a result, it is necessary that the visual attribute *edge.from-node*, which maps to *relationship.from-class*, accept as values concept instances of types *node* and *blob*. Specifically, if $R^*_D(P.A) = I(P')$ for some $P' \in P_D$ then the following should hold:

$$R^*_G(T^{-1}P.A)) = \bigcup_{P'' \in T^{-1}(P')} I(P'')$$

For example, the metaphor of Section 4.3.3 satisfies the above since

$$\begin{aligned} R^*_G(T^{-1}(relationship.from\text{–}class)) &= I(node) \cup I(blob) \\ &= \bigcup_{P \in T^{-1}(class)} I(P) \end{aligned}$$

Otherwise, it would not be possible to choose arbitrary node or blob representations for *entity-class* concept instances.

Second, when there is redundancy in the metaphor, i.e., T_a is not 1-1 and different attributes of the same concept in P_G are mapped to the same attribute of some concept in P_D, the values of the former attributes must be consistent. This can be enforced with constraints in C_G. For example, in the metaphor presented in the previous section, the color and shape attributes of *node* are redundantly used to capture the kind attribute of entity-class. For the metaphor to be correct as defined in Section 4.3.3, the visual model must include the following constraints:

$$\forall n \in \text{node}, \quad \text{color}(n) = \text{‘blue’} \iff \text{shape}(n) = \text{‘square’},$$
$$\forall n \in \text{node}, \quad \text{color}(n) = \text{‘red’} \iff \text{shape}(n) = \text{‘oval’}.$$

Allowing any other combination of shape with color would permit visual schemas with no corresponding data schema. This would prevent the visual model from being used for updates of the data schema, since it would allow conflicting values of the *kind* attribute of the *entity-class*. For example, a visual schema with a blue oval *node* would have no meaning with respect to the metaphor.

Third, the constraints of the visual model should be such that no valid visual schema will map to an invalid data schema, and vice versa. This is ensured through a relationship between the constraints in the data model and those in the visual model. This relationship may be very complex, since one may perform inferences on a given set of constraints to derive additional constraints that are not explicitly specified. For the purposes of this thesis, we take a simple approach and consider only some straightforward sufficient conditions for the consistency of constraints between the two models. Specifically, for two constraints c and c', c *subsumes* c' if the set of schemas that satisfy c is a subset of the set of schemas that satisfy c'. Consider the subset C'_G of the constraints in C_G that are mapped by the metaphor T. (Note that no composition constraint is among them.) If the visual model will be used for retrieval only, then the following should hold:

$$\forall c' \in C_D', \exists c \in C_D, \; c \text{ subsumes } T(c')$$

This is sufficient to ensure that all data schemas have a corresponding visual schema. On the other hand, if the visual model will be used for updates as well, then the following should hold:

$$C_D = \{T(c) \mid c \in C_G'\}$$

This is sufficient to additionally ensure that all visual schemas have a corresponding data schema. Note that the visual model may have additional constraints, those in $C_G - C_G'$, e.g., composition constraints or other constraints that are enforced for presentation purposes.

4.4.2 Metaphor Quality

A metaphor may be correct and nonetheless present information poorly. For example, it is conceivable to have a correct metaphor where T_a is not a function. Such a T_a would map the same visual attribute to more than one data attribute, so that each value of the former corresponds to a combination of values of the latter. We have decided, however, that this introduces a level of visual complexity that is often uncomfortable and would result in confusing visual schemas in many cases. For example, consider the *relationship* concept of the data model in Section 4.2.4, enhanced with a *kind* attribute taking values 'part-of' and 'association'. Following the metaphor of Section 4.3.3, consider mapping the *edge.color* attribute to the combination of the *relationship.kind* and *relationship.card-ratio* attributes. Each of eight colors would map to a given combination of kind and ratio, e.g., T('orange') = ('part-of', 'M:N'). Such a T_a relation is not strictly incorrect, i.e., a well defined mapping between visual and data schemas can still be derived with the desirable properties with respect to information capacity. We believe, however, that such a metaphor would be more difficult for most users to remember than one where two separate visual attributes are mapped to the two data attributes. We thus disallow non-functional T_a's.

Beyond functionality of T_a, other characteristics of *metaphor quality* also affect information presentation. We discuss three such traits that greatly affect metaphors: *information hiding*, which occurs when information is captured by the visual model but is not visible to the user, *visual ambiguity*, which happens when instances of different visual concepts or different values of the same attribute appear identical to the user, and *semantic ambiguity*, which exists when visual attribute values do not suggest the data attribute values they capture. The first two issues are concerned with the visual model alone, while the third regards the metaphor itself. There exist other issues of metaphor quality beyond those mentioned above, including intuitiveness, versatility, and emphasis. These are more difficult to quantify so they are beyond the scope of this thesis.

In general, there are no universal rules about good user interfaces. Nevertheless, we believe there are certain desirable and undesirable characteristics of user interfaces in the context of our own work. We thus conclude the discussion of each of the metaphor quality traits above with comments on the trait's implications, and whether we believe these implications to be good or bad.

4.4.2.1 Hidden Information

Not all information captured by a visual model is visible to the user. This hidden information falls into two categories: transient and structural. Transient hidden information is not visible to the user but can become visible through some manipulation of the visual schema that leaves the underlying data schema unchanged. Consider the visual model described in Section 4.2.4 and the metaphor from Section 4.3.3. The *node* concept

instances have locations that may be anywhere on the plane, thus it is possible for two *nodes* to have the same location. The convention for such cases in a 2-D display system is to make one of the *nodes* invisible or partly visible, conceptually "behind" the other *node*. The *node* that is behind captures information, yet the user cannot see it. If the front *node* is moved, an operation that does not affect the underlying data schema, the back *node* becomes visible. Figure 4.3 demonstrates how one concept instance can be partially or totally hidden by another.

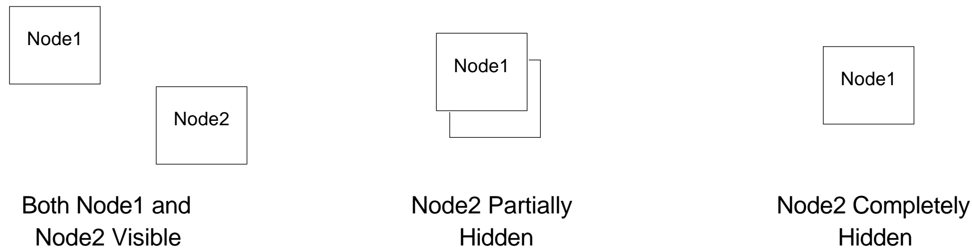

Figure 4.3. An example of transient hidden information.

This example of hidden information depends on the fact that the location attribute is *free*, not part of the metaphor. Freedom of other attributes can also result in transient hidden information. For example, if the size of a *node* were free, the *node* could be hidden by setting its size to zero.

Structural hidden information is information that a visual model captures but does not display. Consider the *edge* concept from the visual model in Section 4.3.1 and the metaphor in Section 4.3.3. It has two attributes, *from-node* and *to-node*, which are not directly shown. Their values are made visible by the constraints that define the location of the *edge*. Consider the result of removing this constraint from the visual model. The location of *edges* would no longer be constrained; an *edge* line could appear anywhere in the visual schema with no relation to the *nodes* specified by its *from-node* and *to-node* attributes. The visual model would still be valid with respect to the data model and metaphor, but it would not be functional for the user. The visual model would still capture the same information, but this information would not be visible to the user. Figure 4.4 demonstrates the appearance of a visual schema with and without the constraints that determine *edge* location. The same metaphor contains another example of structural hidden information. *Edges* are not directed, and as a result it is not possible to distinguish the *edge's from-node* from the *to-node*.

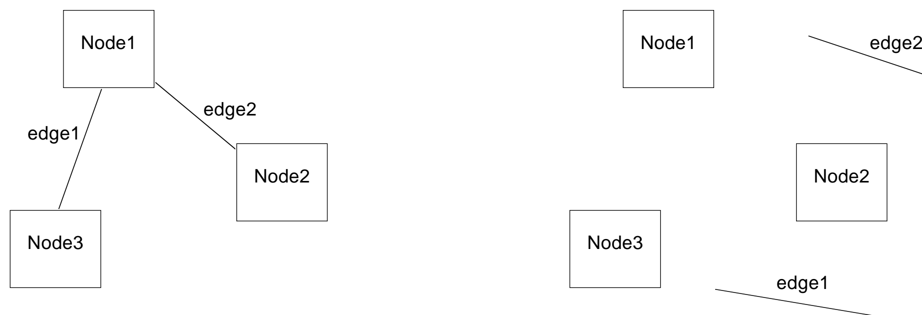

Figure 4.4. An example of structural hidden information.

Structural hidden information can occur whenever the appearance of one attribute is affected by another attribute or constraint. In the above example, the *from-node* attribute is made visible through a constraint that

links it to another attribute, *source-location*.

We believe that structural hidden information should usually be avoided, whereas transient hidden information need not be. In many cases, a model that *cannot* visually display all the information it captures is undesirable. Temporarily invisible information, however, is not a problem as the hidden information can be made visible when necessary. In fact, transient hidden information can be a very useful tool for reducing clutter in a visual schema by hiding infrequently needed information.

4.4.2.3 Visual Ambiguity

Visual ambiguity occurs when a visual model contains two distinct concepts (members of P_G or two distinct values for the same attribute (members of V_G) that are visually indistinguishable. Two items are visually indistinguishable when a user viewing one cannot discern which of the two it is.

Objects with identical appearance are obviously visually indistinguishable. For example, it would be legal to define two different visual concepts with the same attributes and values, and equivalent constraints. Metaphors could be correctly and unambiguously defined (since they depend on symbolic representations of the concepts and their characteristics), but users might be unable to interpret schemas correctly, since instances of the two visual concepts would appear the same. We refer to these cases as strict visual ambiguity. A slightly less strict form of visual ambiguity occurs in cases where attributes of two different concepts have different names, but the same values and constraints.

Another kind of visual ambiguity occurs when two concepts or attribute values appear very similar, though not identical. Concepts or attributes with a similar appearance may be visually indistinguishable, depending in part on the degree of similarity and the visual acuity of the viewer. For example, if two polygons of 15 and 17 sides are mapped to two different values of a data attribute, Figure 4.5 shows that users might not be able to correctly distinguish between the two values. Another example would be two concepts with different attributes and values but the same appearance to the user.

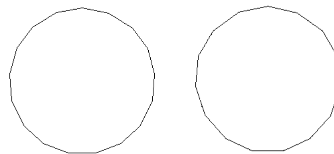

Figure 4.5. Polygons of 15 and 17 sides.

Visual ambiguity is, in general, undesirable, because it results in schemas of a visual model that may not be correctly interpreted by a user. Strict visual ambiguity is straightforward to detect by testing the Q_G and R_G functions, and the constraints affecting the concepts in question. Non-strict ambiguity is much harder to define formally, let alone detect; there may not exist universal similarity (as opposed to equality) measures. Investigating possible definitions and similarity detection algorithms is part of our future work.

4.4.2.3 Semantic Ambiguity

Semantic ambiguity occurs when the appearance of a visual attribute value does not bring to mind the data attribute value with which it corresponds. The degree of ambiguity depends upon the memory of the user, the range of data attribute values, the type of visual attribute, and the choice of visual attribute values. For example,

using randomly assigned colors to represent values between 1 and 500 would be problematic for any user lacking an perfect memory (if the values are of interest to the user). Using colors ordered and spaced by their place in the spectrum would be better, giving the user a feel for the magnitude of different values. Using Arabic numerals to visually represent these values would be the most precise (though possibly less effective for giving a quick impression of the value). While improving human memory is beyond the scope of this thesis, we can offer visual attribute and value choice guidelines to reduce semantic ambiguity.

In general, any visual attribute type can be used for representing a data attribute with a small value range. For example, it would not be difficult for a user to learn associations between a small number of shapes, colors, or patterns, and their corresponding data model values. Precisely capturing values with a larger range, however, requires visual attributes with inherent meaning; the visual value must in some way suggest the data value. Attribute types may be divided into two categories with respect to inherent meaning. Text and pictures can have much inherent meaning as long as the viewer shares a linguistic or cultural context with the creator. Some attribute values are almost universal in their inherent meaning, e.g., the image of a human face carries the meaning “a human face” to most humans who see it. Others require more cultural context, for example, a picture formed of a red octagon with the word “STOP” in the middle has an immediate association for people who have experience with roads in certain countries. Shape, color, pattern, size, and location have less inherent meaning, and what meaning they have is limited to more narrow contexts. For example, a red colored light means “stop” in the context of driving, but it also means “on” in the context of electric kitchen ovens and toasters. If a precise representation is not needed, the limited inherent meaning of these attributes can be useful. For example, consider a metaphor that associates the spectrum of colors to a large range of temperatures. While specific values would be hard to determine, the user could easily make comparisons and determine general magnitudes of values.

As a result, in most cases text and pictures should be used to represent values with large ranges. In some cases, when a general impression of the value is needed instead of the exact value, other attributes can be used. Values with a smaller range, such as entity-class kind from Section 4.2.2, may be represented by any type of attribute.

4.5 Combining and Mixing Metaphors

Different metaphors have different characteristics: emphasis, space efficiency, intuitiveness, and versatility. There is no single metaphor that is best for all schemas and all situations. We believe that a schema visualization tool should support a variety of metaphors and associated visual models. Users will be able to choose among them so that the same data schema may be viewed as different visual schemas, each suitable for different circumstances.

The use of different metaphors may be taken one step further, by allowing the use of different metaphor correspondences for different parts of the same visual schema. This is faithful to the definition of metaphors in Section 4.3.1, which allows correspondences between one data model concept and several visual model concepts. Such metaphors may originally be defined this way, or may be defined as a combination of two simpler metaphors. This involves combining their visual models into a single, unified model, and combining the metaphors to map from that model. The following section presents some example metaphors that will be used to demonstrate the formal specification of metaphor combination.

Model	Concept (P)	Attribute ($P.A$)	Attribute Values ($R^*(P.A)$)
G_1	node	shape	{oval}
		location	plane-points
		size	{100 pixels}
		color	{white}
		label-text	text
		label-color	{blue,red}
	edge	source-location	plane-points
		dest-location	plane-points
		color	{red, orange, magenta, green}
		from-node	I(node)
		to-node	I(node)
		label-text	text
		label-color	{black}
G_2	node	shape	{rectangle}
		location	plane-points
		size	{100 pixels}
		color	{yellow, brown}
		label-text	text
		label-color	{black}
	arrangement	label-text	text
		label-color	{red, orange, magenta, green}
		label-location	plane-points
		parent-node	I(node)
		child-node	I(node)

Table 4.6. Two Visual Models.

4.5.1 Example Visual Metaphors

Consider the data model D of *entity-classes* and *relationships* described in Section 4.2.4. Also consider the visual models, G_1 and G_2, described in table 4.6. Visual model G_1 is similar to G described in Section 4.2.4. Visual model G_1 has *nodes* that are rectangles, and instead of using *edges* to represent a connection between two *nodes*, it uses *arrangements*. *Arrangements* are formed from a *text-display* construct and two *nodes*, a parent-node and child-node. The existence of an *arrangement* affects the location of the child-node. The specific physical arrangement is defined by a set of constraints which require *node* placement similar to a textual outline, where subpoints appear below and indented to the right of the main points. It should be noted that G_2 is a visual model not for general directed graphs but only for trees, as any child *node* with multiple parents would have conflicting constraints on its location. These models are accompanied by several composition constraints, which are of no particular interest and are therefore not shown.

For each visual model, we define a metaphor. The two metaphors are shown in Tables 4.7 and 4.8. For brevity, the part of the metaphor that corresponds to the T_v function is not included. Visual metaphor

$T_1 : G_1 \rightarrow D$ is similar to the metaphor described in Section 4.3.1, except that it does not include the *blob* concept, and *entity-class.kind* is represented by *node.label-color*. Visual metaphor $T_2 : G_2 \rightarrow D$ is different in that *entity-class.kind* is captured by *node.color*, and that *relationships* are expressed as physical arrangements of the related *nodes* (as described earlier). Figure 4.6 gives examples of a simple schema displayed using each of the metaphors. This example, drawn from the Cupid simulation model (described in Chapter 6), shows a case where the outline metaphor is more compact than the graph metaphor.

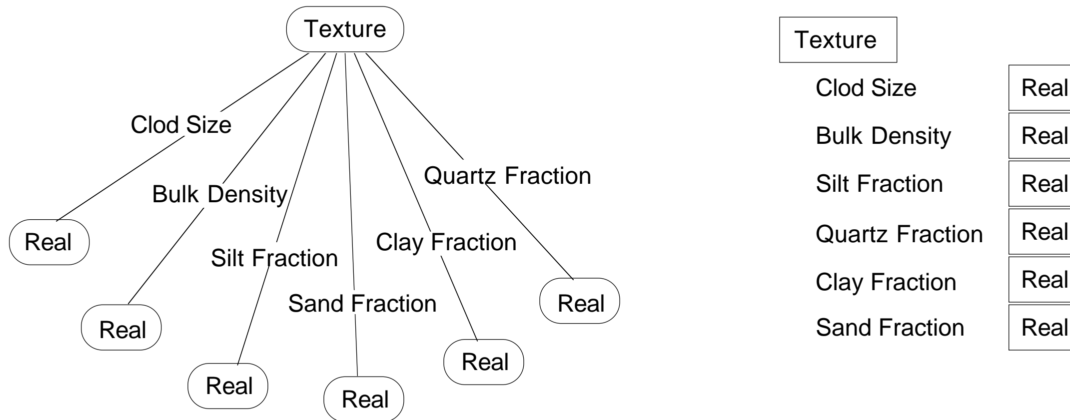

Figure 4.6. An example schema displayed using each of the two metaphors.

G_1	$T_1(x)$
node	entity-class
node.label-text	entity-class.name
node.label-color	entity-class.kind
edge	relationship
edge.label-text	relationship.name
edge.from-node	relationship.from-class
edge.to-node	relationship.to-class
edge.label-color	relationship.card-ratio

Table 4.7. One sample metaphor.

G_2	$T_2(x)$
node	entity-class
node.label-text	entity-class.name
node.color	entity-class.kind
arrangement	relationship
arrangement.label-text	relationship.name
arrangement.parent-node	relationship.from-class
arrangement.child-node	relationship.to-class
arrangement.label-color	relationship.card-ratio

Table 4.8. Another sample metaphor.

4.5.2 Combining Visual Models and Metaphors

In order to use different metaphors for different parts of a schema, the visual models associated with these metaphors must be combined into a single model. In addition, the metaphors themselves must be combined to form a unified metaphor, mapping from the combined visual model to the data model. Combining the visual models ensures that the concepts from different models may be used together, and that the metaphors themselves may be combined.

There are several abstractions that could be used to model the combination of visual models and metaphors, any of which results in a valid, unambiguous, and usable metaphor and schema mappings. These abstractions differ in the level of mixing that they permit of the visual models and metaphor functions. In this subsection, we discuss an abstraction that allows mixing at all levels. The subsequent subsections describe correctness and quality issues involved in mixing metaphors based on that abstraction.

For the visual models $G_1 = < P_{G_1}, A_{G_1}, V_{G_1}, Q_{G_1}, R_{G_1}, C_{G_1} >$, and $G_2 = < P_{G_2}, A_{G_2}, V_{G_2}, Q_{G_2}, R_{G_2}, C_{G_2} >$, consider their combination $G = < P_G, A_G, V_G, Q_G, R_G, C_G >$. By definition, for any concept P that is common to both visual models, the equality $Q_{G_1}(P) = Q_{G_2}(P)$ holds, i.e., the same concept has the same attributes in all visual models that include it. The elements of the visual models are combined as follows:

$$\begin{aligned} P_G &= P_{G_1} \cup P_{G_2} \\ A_G &= A_{G_1} \cup A_{G_2} \\ V_G &= V_{G_1} \cup V_{G_2} \\ C_G &= C_{G_1} \cup C_{G_2} \end{aligned}$$

Note that, based on the naming convention established in Definition 1, the above equations imply that:

$$\begin{aligned} &\forall P \in P_G,\ Q_G(P) = Q_{G_1}(P) \cup Q_{G_2}(P) \\ &\forall P.A \in A_G,\ R_G(P.A) = R_{G_1}(P.A) \cup R_{G_2}(P.A) \end{aligned}$$

Given a combined visual model, two metaphors $T_1 = T_{p1} \cup T_{a1} \cup T_{v1}$ and $T_2 = T_{p2} \cup T_{a2} \cup T_{v2}$ may be combined to form a unified metaphor $T = T_p \cup T_a \cup T_v$ where:

$$\begin{aligned} T_p &= T_{p_1} \cup T_{p_2} \\ T_a &= T_{a_1} \cup T_{a_2} \\ T_v &= T_{v_1} \cup T_{v_2} \end{aligned}$$

4.5.3 Correct and Good Mixing of Metaphors

The result of combining two metaphors using the process shown in the previous section must satisfy Definition 3 in order for it to be a metaphor itself. Assume that T_1 and T_2 are correct metaphors with respect to either viewing or updating data schemas. Then, T_1 and T_2 are onto functions with possible additional properties of totality and/or 1-1ness. When taking the union of T_{x1} and T_{x2} (for $x \in \{p, a, v\}$), totality and onto-ness can never be lost. 1-1ness may be lost when unioning, but it is unrelated to correctness and only affects redundancy and choice in a metaphor (Section 4.3.2), so it does not present a problem. Functionality, however, is necessary

for correctness and may be lost when unioning. In that case, T is not a correct metaphor, implying that the original metaphors are not combinable. To correctly combine T_1 and T_2, the resulting T_p, T_a, and T_v must be functions.

The combined metaphor must also satisfy the criteria from Section 4.4.1. In addition, the new set of constraints established by unioning the constraints of the two original visual models must contain no contradictions and should not exclude any visual schema that was valid in the two original visual models. If any of the above does not hold, then the original metaphors are not combinable.

We should emphasize once again that one could use a different abstraction from that described in Section 4.5.2 to combine metaphors. Such an abstraction would possibly allow different pairs of metaphors to become combinable. We have chosen the above abstraction for its simplicity and because it captures several desirable metaphor combinations.

4.5.4 Example Metaphor Combination

Consider the example metaphors from the previous section. When the two are combined, the metaphor will appear as shown in Table 4.9. Where both original metaphors are the same, such as the mapping of *entity-class* or *entity-class.name*, the combined metaphor is the same. Where the original metaphors diverge, the combined metaphor either offers choice (as in the case of *relationships*) or redundancy (as with *entity-class.kind*).

The unified visual model undergoes similar changes, as shown in Table 4.10.

x	$T(x)$
node	entity-class
node.label-text	entity-class.name
node.label-color	entity-class.kind
node.color	entity-class.kind
edge	relationship
arrangement	relationship
edge.label-text	relationship.name
arrangement.label-text	relationship.name
edge.from-node	relationship.from-class
arrangement.parent-node	relationship.from-class
edge.to-node	relationship.to-class
arrangement.child-node	relationship.to-class
edge.label-color	relationship.card-ratio
arrangement.label-color	relationship.card-ratio

Table 4.9. A Combination of Two Metaphors.

Concept (P)	Attribute ($P.A$)	Attribute Values ($R^*(P.A)$)
node	shape	{oval, rectangle}
	location	plane-points
	size	{100 pixels}
	color	{yellow, brown, white}
	label-text	text
	label-color	{blue,red, black}
edge	source-location	plane-points
	dest-location	plane-points
	color	{red, orange, magenta, green}
	from-node	I(node)
	to-node	I(node)
	label-text	text
	label-color	{black}
arrangement	label	text
	label-color	{red, orange, magenta, green}
	label-location	plane-points
	parent-node	I(node)
	child-node	I(node)

Table 4.10. A Unified Visual Model

Nodes existed in both of the original models, yet they had different shapes. In the combined model, a choice of shape exists. Figure 4.7 gives an example of a schema displayed using the mixed metaphor. Note how *node* shape is either oval or rectangular, and how *relationships* may be displayed either using an *edge* or an *arrangement*.

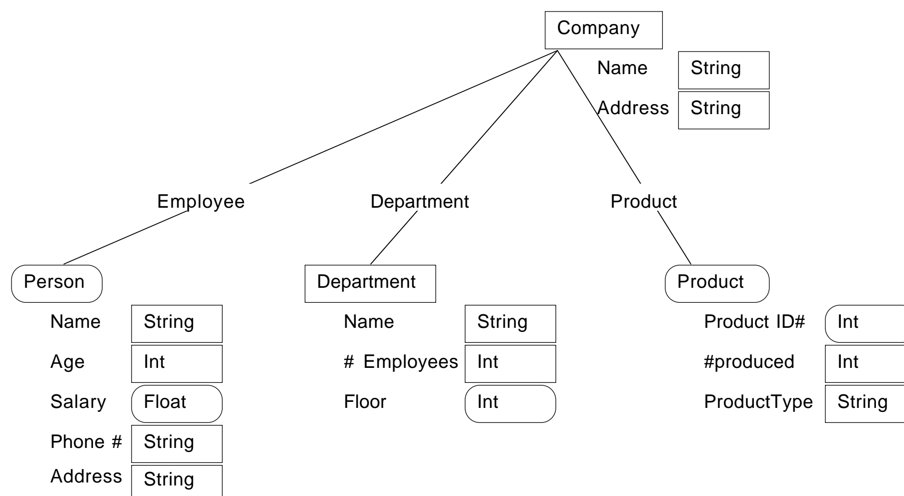

Figure 4.7. An example of a schema displayed using the combined metaphor.

4.6 Visually Capturing Additional Information

The definition of models and metaphors allows the visual model to have greater information capacity than the data model. Specifically, there may be more concepts in the visual model than in the data model, the visual model concepts used in the metaphor may have more attributes than their corresponding data model concepts, and the range of visual attribute values may be greater than that of their corresponding data model attributes. It is possible to define the visual model to have the same information capacity as the data model, but often extra information capacity is valuable. Surplus information capacity may be used in two ways. One is to enrich the metaphor. For example, the T_p , T_a and T_v functions may be many-to-one, allowing redundancy and choice in representing information. The other use of extra information capacity, and the subject of this section, is to capture information outside of the data schema. This information may be divided into two categories: presentation and personal model information. The *personal model* is a superset of the data model, additionally containing information that is part of the user's conception but not captured by the database. Presentation information is any visual information not mapped by the metaphor. These two kinds of information will be discussed in depth in this section.

4.6.1 Presentation Information

Presentation information is visual information that is not mapped by the metaphor. For example, the locations of nodes in a directed graph are usually chosen for purely aesthetic reasons, so location would not be part of the metaphor. While not capturing any part of the data schema, this information is important as it can affect the readability of a presentation, and it may express meaning for the user that is difficult or impossible to capture in the data model. There are many ways to lay out a directed graph, all having the same meaning, but some are much more readable than others.

Presentation information is captured by those surplus visual model concepts and attributes that are not in the domain of the metaphor, and by the attribute value and concept choices that are part of the metaphor. For an example of the use of extra concepts, consider the visual model and metaphor from Section 4.3.1 supplemented with the following concept:

Concept (P)	Attribute ($P.A$)	Attribute Values ($R^*(P.A)$)
rect	shape	{ rectangle }
	location	plane-points
	size	integer $\times$ integer pixels
	background-color	{ white }
	border-color	{ black }
	border-width	{ 2 } pixels

Table 4.11. The Rect concept.

This *rect* is a black bordered rectangle of any given size and location. Not part of the metaphor's domain, it can be used freely without affecting the meaning of a visual schema. For example, a *rect* could be placed around the entire schema to give it a border and improve its appearance.

An example of surplus attributes may be found in the same metaphor: *node-location* is not specified by the mapping and can be freely specified as mentioned above. Similarly, the possibility of representing 1:1 relationships as either 'green' or 'orange' *edges* allows the user aesthetic leeway without changing the meaning of the visual schema.

Another means of capturing presentation information is through choice of visual model concepts. Different concepts may have very different appearances, affecting the aesthetics of the schema. For example, consider the combined metaphor described in Section 4.5.2. It maps two visual concepts, *edge* and *arrangement*, to *relationships*. These visual concepts have very different appearances and would be suitable in different situations.

4.6.2 Personal Data Model Information

Databases are commonly used for holding information about real-world items. Data schemas describe the organization of these items in as much detail as allowed by the data model. Frequently, however, there exists other organizational information about these items that might be helpful to the user, but is not or cannot be captured by the data schema.

We introduce the notion of a *personal data model* to capture the organization of the database from the user's viewpoint. This is no different from any other data model, except for the fact that each user may have a different personal data model (while there is a single system data model) and also that each personal data model must be an extension of the system data model. Accordingly, *extended visual metaphors* may be defined from a visual model to personal data models to capture the full range of characteristics of personal data schemas, as shown in Figure 4.8. Such an extended metaphor would map visual model concepts, attributes, and values not used by the regular metaphor (i.e., its excess information capacity) to corresponding constructs of the personal data model that are not part of the system data model. Figure 4.8 demonstrates the distinction between Personal and Aesthetic information: Personal information may be preserved between different visual models, and Aesthetic information can only be preserved within a single visual model.

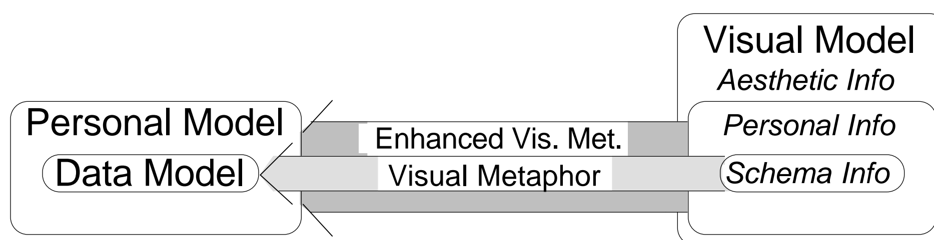

Figure 4.8. The personal data model and metaphor as extensions of the data model and visual metaphor

As an example of a personal data model and its corresponding personal metaphor, consider the data model of Section 4.2.4, whose concepts are *entity-class* and *relationship*. Like other object-oriented/semantic data models, this model does not allow higher-level groupings of *entity-classes* or *relationships*, an ability that could be very useful to a user. For example, a schema combining data from an experiment could have *entity-classes* grouped by their roles within the experiment (e.g., input versus output), whether their contents are considered accurate, and their significance to the user. Multiple simultaneous orthogonal groupings may be captured this way, with an *entity-class* belonging to several different groups for different reasons. This may be achieved by allowing a user to extend the above data model so that groups can be captured. The visual

metaphor from Section 4.3.1 may also be modified to represent group information to the user. In the original metaphor, *node* size, label-color, shape, and location are attributes that are not part of the metaphor mapping. If the visual model were defined to allow these attributes a greater range of values, they could be used by the personal metaphor to enhance the information that is captured visually. Figures 4.9, 4.10, and 4.11 give examples of an unmodified visual schema, grouping by location, and grouping by shape and location, respectively.

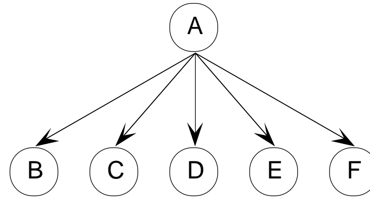

Figure 4.9. A directed graph.

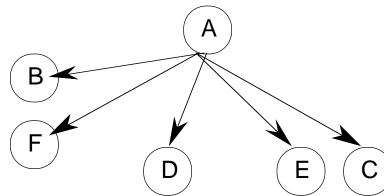

Figure 4.10. Grouping by location.

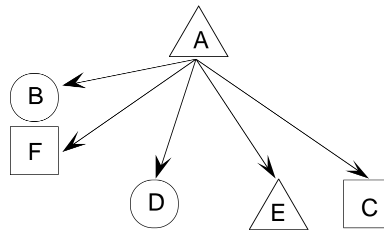

Figure 4.11. Grouping by location and shape.

4.7 Managing Layout when Concept Instances Change Size

4.7.1 Introduction

Whenever visual concept instances change in size, problems of visual schema layout can occur: when instances grow, undesirable overlap may result, and when instances shrink, the schema can become overly sparse. Changes in size can happen when mixed metaphors are used, or any time the size of a visual concept instance is variable. For example, if a single data model concept is mapped to by two visual concepts of widely varying sizes, changing the metaphor for a concept instance can cause a visual schema to become either more crowded or more sparse. Consider a new visual model G_3 shown in Table 4.11, and mixed metaphor (not shown) that maps from G_3 to the data model D from section 4.2.4. This visual model and metaphor represent *entity-classes* as either *small-nodes* or *big-nodes,* and *relationships* as *edges*. A *small-node* is an oval just large enough to contain a single label, which has the text “P” or “C” indicating whether the *entity-class* is

primitive or compound. A *large-node* is a larger rectangle holding labels for the name of the *entity-class,* its kind (represented as the strings "Primitive" and "Compound"), and a comment field that displays the personal information of an annotation describing the *entity-class.*

Model	Concept (P)	Attribute $(P.A)$	Attribute Values$(R^*(P.A))$
G_3	small-node	shape	{oval}
		location	plane-points
		size	{12 pixels}
		color	{white}
		label-text	{"P", "C"}
		label-color	{black}
	big-node	shape	{rectangle}
		location	plane-points
		color	{white}
		label1-text	text
		label1-color	{black}
		label2-text	{"Primitive", "Compound"}
		label3-text	text
	edge	source-location	plane-points
		dest-location	plane-points
		color	{red, orange, magenta, green}
		from-node	I({small-node, big-node})
		to-node	I({small-node, big-node})
		label-text	text
		label-color	{black}

Table 4.12. Selected attributes of the concepts in visual model G_3.

An example schema in model G_3 is shown in Figure 4.12. Note that if the visual representation any of the entity-classes is changed from small-node to big-node, it is clear that overlap with other nodes might result. Such overlap would be undesirable, hiding other nodes and obscuring the structure of the schema. Undesirable overlap is sufficiently common that we provide a pair of general mechanisms for managing it. These mechanisms are necessary because the formalism does not capture behavior: it is possible to indicate that overlap is illegal using constraints, but it is not possible to specify what actions should take place when overlap occurs. In addition to overlap, another problem that results from concept instances changing size. When large concept instances get smaller, the schema becomes more sparse, and space is less well utilized. Ideally, any solution for managing schemas that are too crowded will also help with schemas that are too sparse. The following subsections describe two heuristics for managing overlap when concept instances become larger, and utilizing empty space when concept instances become smaller.

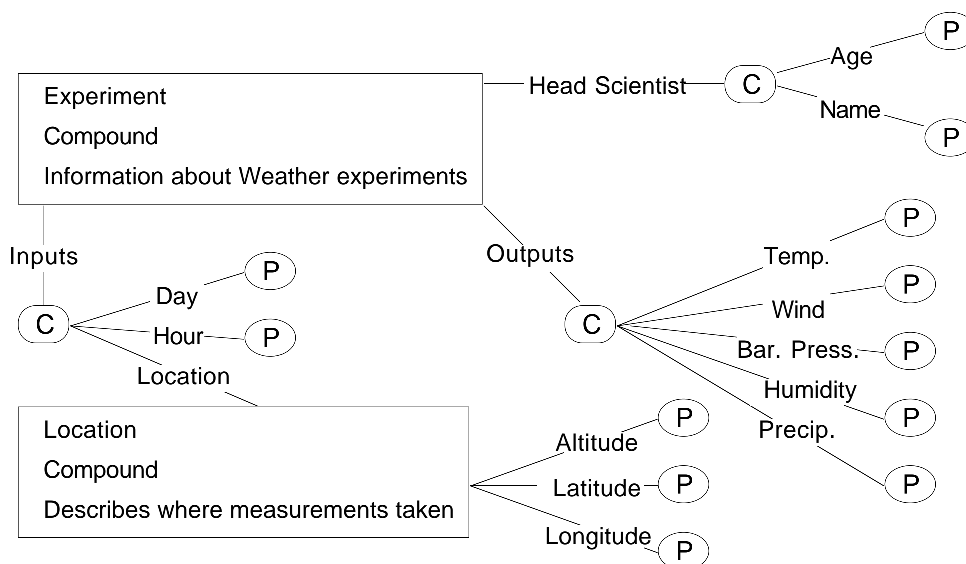

Figure 4.12. A sample schema in model G_3 .

We have chosen to address these problems using approaches that are partially automated, i.e. they rely upon a certain degree of input from the user. We have found that users rely upon the relative locations of concept instances in a schema, so we want an approach that allows users to specify the layout of a schema and that keeps the overall shape of the schema similar as concept instances change in size. Other work exists on fully automatic layout of trees and directed graphs [ET89, Mess88, Moe90], but this work has two limitations. The approach to trees considers user-specified information, but it is not sufficiently general to apply to arbitrary visual models, and the work on graphs produces unstable layouts, i.e., a small change in a graph can result in a large change in the layout. Unstable layout algorithms lose the important user-specified location information.

4.7.2 A simple approach to manage overlap: Prevention & Plowing

The simplest approach to managing overlap is to prevent it. Constraints can specify between what concepts is overlap forbidden, and whenever an operation would result in a schema with illegal overlap, that operation in not permitted. This solution is satisfactory for some operations, such as movement, but is less adequate for others. For example, when the label of a *big-node* is lengthened, making the rectangle wider, it would be unpleasant to have the system disallow the change because enough space is not available. This approach can be improved by pushing other things out of the way using a “plowing” algorithm. Plowing, described generally by Ousterhout [Ous84], is an approach where overlapping items are pushed out of the way, just as a plow pushes soil. We have implemented plowing as follows:

```
- Consider a concept instance that has increased in size.
- Find all other instances that overlap the instance and should not.
- Divide the overlapping instances into 4 groups, those to move up,
  down, right, or left, depending on the minimal amount of movement
  required to eliminate the overlap by moving them in those
  directions.
```

```
- Move all members of the "up" group upward the same amount,
  sufficient to eliminate the overlap for all of them.  If anything
  overlaps a member of the "up" group, move the new overlapping item
  upward the same amount.
- Repeat the previous step for the down, left, and right groups.
```

This approach may be extended to handle over-sparse graphs as well. Just as certain operations can result in overlap and concept instances being pushed from their locations via a plowing algorithm, when the same operations make space available, neighbors of that space should be pulled in using plowing in reverse. The reverse-plowing algorithm works as follows:

```
- Consider a concept instance that has decreased in size, and its
  reduction in size horizontally and vertically.
- Find 4 groups of neighbors based on the original instance size:
  those within one half of the original width left and right, and
  those within half of the original height up and down.
- If the left group has neighbors within the same distance to its
  left, add those neighbors to the left group.  Repeat this step
  recursively.
- Repeat the previous step for the top, bottom, and right groups.
- Given the reduction in size horizontally, move the left group that
  same amount towards the shrinking instance.  If any overlap
  occurs, undo the move for the overlapping item and any member of
  the left group to the left of it.
- Repeat the previous step for the right, top and bottom groups.
```

This algorithm is overzealous: it may pull back more instances than plowing will push. As a result, plowing and reverse plowing may make the schema more compact over time, changing the relative locations of different parts of the schema. For example, Figures 4.13 through 4.15 show a *small-node* changing to a *big-node* and back again. In some situations these problems are not serious, and the combination of prevention for some operations and plowing/reverse-plowing for others is sufficient.[7] In many situations, however, the limitations of the prevention/plowing approach would be unacceptable. For example, in many cases the user would like the overall shape of the schema to stay the same as instances change in size. The next subsection describes a more advanced solution that can preserve user specified layout as concept instances get larger and smaller.

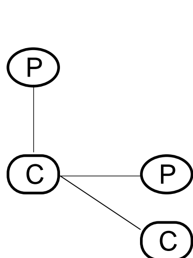

Figure 4.13.
A simple schema.

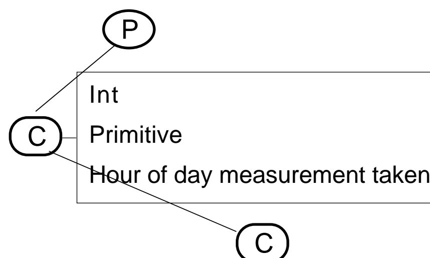

Figure 4.14.
small-node to *big-node*.

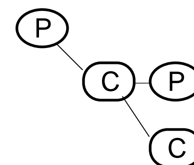

Figure 4.15.
Back to *small-node*.

[7]Deciding which operations should be disallowed, and which should result in plowing is a subject for user evaluation, and will not be considered further here.

4.7.3 A more advanced approach: move towards safe locations

In some situations, the user specifies schema layout and would like the schema to return to that layout if concept instances change in size, then return to their original state. This subsection describes an approach that uses user-specified layout information to guide automatic changes to the schema. The central idea of this approach is that the system should record certain user-specified non-overlapping layouts. Concept instance's locations in these layouts are called "safe", because when everything is in its safe location, there is no overlap. In general, when overlap occurs, concept instances are moved towards their safe locations until there is no overlap. This approach relies upon the following assumptions:

- The cause of overlap and sparsity is that one or more concepts may not overlap, that they have representations of varying size, and that for each type the representations may be ordered by size.
- The user wishes to move between several layouts of the schema, some more compact than others, but all in the same general area, and all having the same general shape.
- The original layout is user specified.

This approach to layout management is best introduced through an example, given in the following subsection.

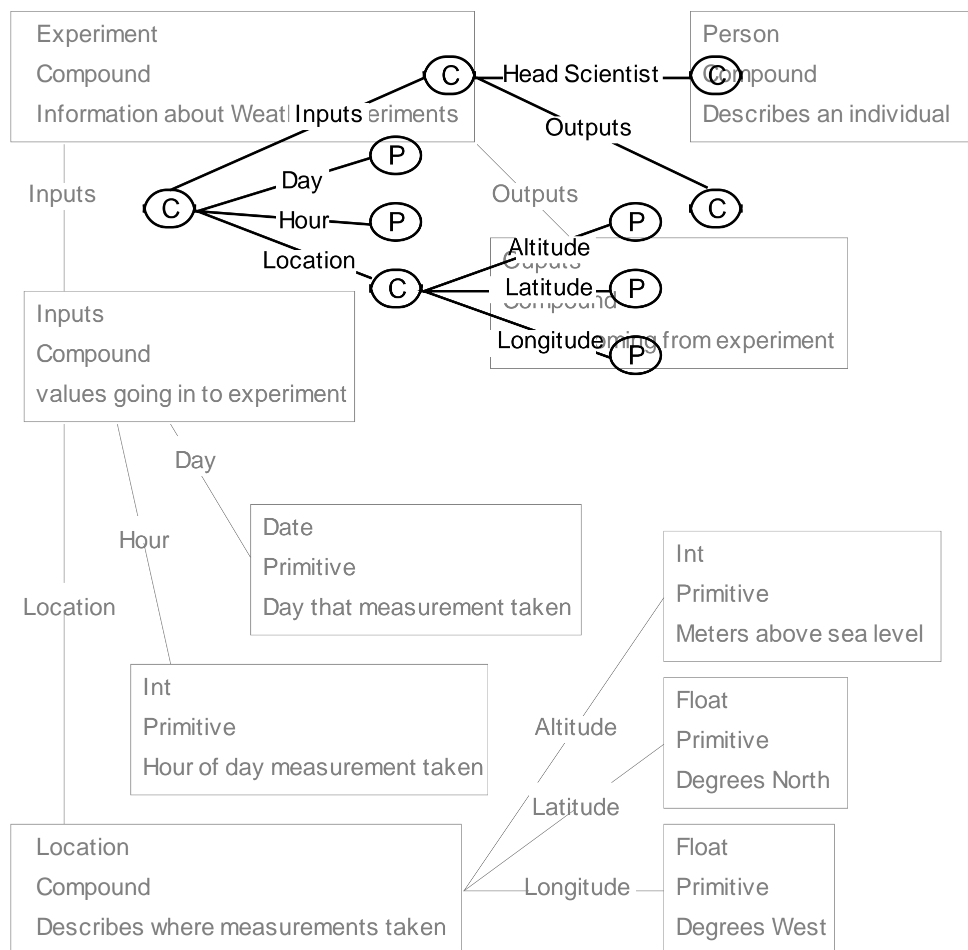

Figure 4.16. The big schema layout and small schema layout, overlayed.

4.7.3.1 A simple example

Consider a simplified version of the schema in Figure 4.12. Through some mechanism (to be explained later), the user specifies a non-overlapping layout for the entire schema when all the *entity-classes* are represented as *small-nodes* (referred to as the "small schema layout"), and another such layout when all the *entity-classes* are shown as *big-nodes* (the "big schema layout"). Figure 4.16 shows these two layouts, one on top of the other. These two layouts are recorded by the system. For each concept instance, the location it occupied in the big schema layout will be called its big-safe location, and its location in the small schema layout will be called its small-safe location. The word "safe" is used in these terms because in the two extreme cases, of the biggest schema and the smallest schema, those locations were safe with respect to overlap with other concept instances. No two concept instances are allowed to have overlapping big-safe or small-safe locations (though big-safe locations may overlap with small-safe locations).

This layout information is used as follows. Consider starting from the small schema. Whenever a concept instance changes representation from small to big, if it causes overlap in its new location:

```
- The items it overlaps are moved on a straight line from their current location
  toward their big-safe locations until they get there, or until they no longer
  overlap anything, whichever comes first.
- If they get to their big-safe locations and still overlap other concept
  instances, those others are moved towards their big-safe locations.
- This process is repeated until there is no more overlap.
```

For example, consider the simple schema in Figure 4.17. If the *small-node* representing hours were changed to a *big-node,* it would cause its neighboring *small-nodes* to be moved downward toward their big-safe locations, as shown in Figure 4.18. If subsequently the *small-node* representing inputs were changed to a *big-node,* then the *big-node* representing hours would be moved towards its big-safe location. This would cause it to overlap with the two *small-nodes,* causing them to be moved towards their big-safe locations. One of the two *small-nodes* can only be moved a short distance before it reaches its big-safe location, causing the *big-node* to be moved further. The resulting schema is shown in Figure 4.19. If the other two *small-nodes* were expanded, then the schema would be close to its original big-layout.

Available space is managed using a similar approach. Whenever a concept instance changes representation from a *big-node* to a *small-node,* it is moved toward its small-safe location until it gets there or finds another node in the way, at which point it stops. This process frees space, so any neighboring nodes are moved (if necessary) toward their own small-safe locations.

This example explained the gist of the move-toward-safe-locations approach. The next two sections describe in more detail how the safe locations are defined, and how the layout algorithm uses safe locations when overlap and sparsity is detected.

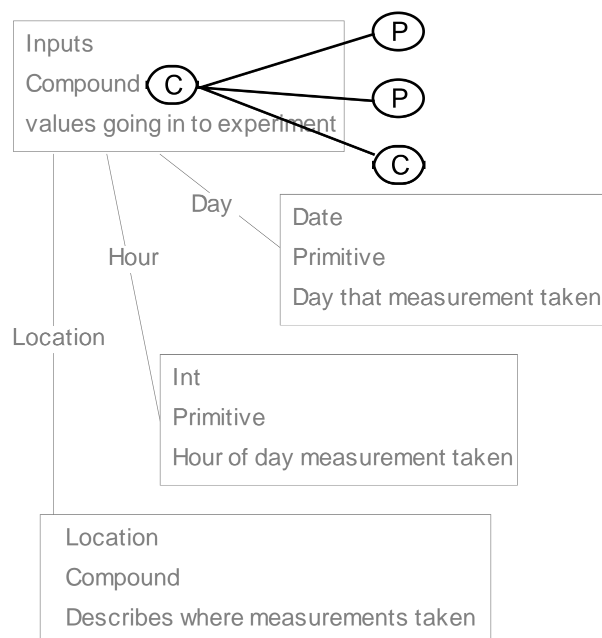

Figure 4.17. A simpler version of the schema in Figure 4.16.

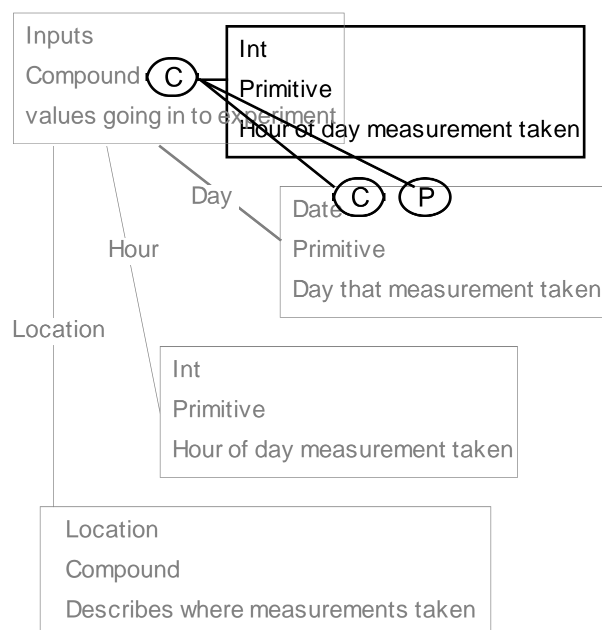

Figure 4.18. The same schema with Hour as a *big-node*.

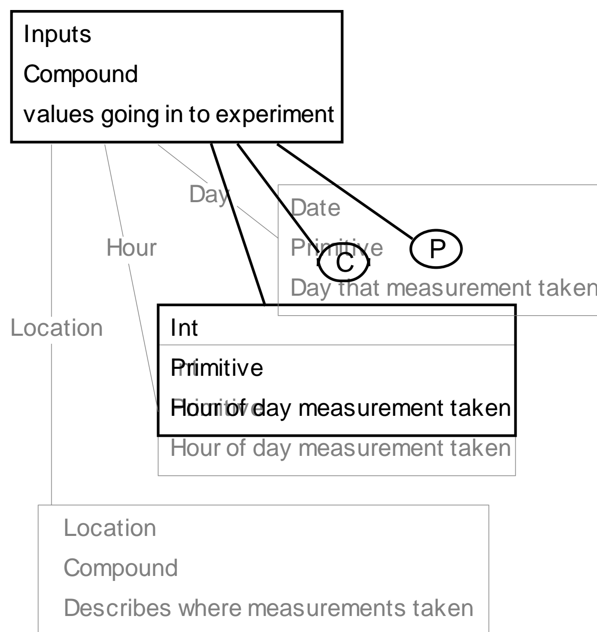

Figure 4.19. The same schema with Inputs as a *big-node.*

4.7.3.2 Defining the safe location

The example in Section 4.7.3.1 had two safe layouts, one for each representation of the *entity-class.* In the general case, there will be many safe layouts, and for all concepts that are not allowed to overlap, each of their various representations must be associated with a safe layout. For example, if a concept had five different representations, the user might want five safe layouts between which the schema would move. These layouts are prioritized; in the above example the concept instances in their large-safe location took precedence over concept instances in their small-safe locations. The priority should be given in descending order from largest representation to smallest, because a smaller representation can always fit inside the safe location of a larger representation. Thus, the system maintains an ordered list of consistent layouts.

Given the potentially large number of safe layouts, it would be infeasible to ask the user for every safe location each time a new concept instance is added to the schema. In addition, our users have found it unsatisfactory to be prevented from adding a new instance to an unsafe location (i.e. where it currently overlaps the safe locations of other instances at that level). As a result, we have decided to allow safe locations to be in a state called unspecified. All safe locations for a concept instance are unspecified until the instance is located such that its current footprint does not overlap any safe locations for its level. When an instance has an unspecified safe location, the layout algorithm is reduced to pushing it using a plowing algorithm.

The safe locations for a concept instance are defined as follows:

```
- Initially, all of a concept instances safe locations are undefined.
- When a concept instance is created, moved, stretched, or changed to a new
   representation, the system checks the safe layout associated with the new
   representation/location.  If the new footprint of the concept instance fits in
   the safe layout for the representation, then the safe location for that
   representation is defined as the new footprint, otherwise the safe location
   remains what it was before.
```

These rules ensure that no two concept instances have the same safe location in the same safe layout. The next subsection describes the rules for managing overlap given safe locations in different layouts, and the possibility of unspecified safe locations.

4.7.3.3 Managing layout after overlap or freed space

Given the rules specified in the previous subsection, the system will maintain an ordered list of safe layouts, and each concept instance may have a safe location for each of its different representations in those layouts (though the location may be unspecified instead). This section describes the rules for managing overlap between concept instances when a change to the schema results in overlap. When overlap occurs between two concept instances:

```
- If one concept instance is in a representation of a higher priority than the
  other, move the other to its safe location in the higher priority layout.  If
  its location at that priority is unspecified, or if it is too big to fit in that
  location, push it using the plowing algorithm described above.
- If the concept instances are in representations associated with the same safe
  layout, move one towards its safe locations in that layout.  If one or both have
  unspecified safe locations, push one using a plowing algorithm.
```

Whenever a concept instance changes to a smaller representation:

```
- If it has a safe location for that representation, move toward that location
  until it gets there, or until it will overlap something, and stop.
```

When space is freed up as a result of some change to the schema:

```
- Move any neighbors (those instances within some fixed distance of the freed
  space) toward their lowest priority (smallest) safe locations until they either
  get there or will overlap something, and stop.
```

Thus, whenever overlap occurs, concept instances are moved toward their location in a more spacious layout, and whenever space is freed, they are moved toward their locations in a more compact layout.

4.7.4 Layout Summary

This section has described two approaches to managing layout for schemas whose members change in size. If the sizes of different representations can be ordered, then the more advanced heuristic works very well in returning the schema to a prespecified layout after a number of changes. It also keeps the shape of the schema similar between layouts. If no such ordering exists, then the advanced heuristic does not handle mixed representation schemas very well, and the simpler approach would be sufficient.

4.8 Summary

In this chapter, we have presented a formalism for visual metaphors and described how it may be used to improve the visual presentation of data schemas. This formalism allows high level description of the correspondence between data and visual models. This description allows simpler definition of metaphors, easier evaluation and comparison of metaphors, and combination of different metaphors. The formalism can help improve schema visualizations in the many roles they play. Currently, the formalism has been implemented as an editing tool for schemas of arbitrary data models, visual models, and metaphors. This tool is described in detail in Chapter 5.