Chapter 5
An Overview of the OPOSSUM DTSM

5.1 Introduction

In order to demonstrate the power of the formalism discussed in the previous chapter, a Desk-Top Schema Manager called OPOSSUM (**O**btaining **P**resentations **O**f **S**emantic **S**chemas **U**sing **M**etaphors) has been implemented. It provides all of the DTSM capabilities described in Chapter 3: creation of schemas in many data models through direct manipulation of visual schemas, creation and modification of models and metaphors, addition of user-specific personal information to schemas, accommodation of varying aesthetics between different users, and support for choice of visual style at many granularities. This chapter describes OPOSSUM and how it achieves these goals.

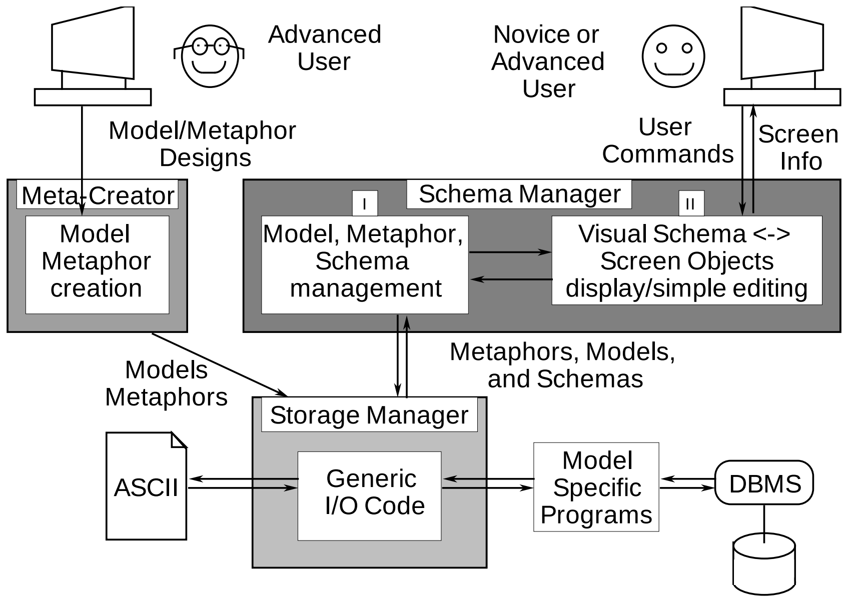


Figure 5.1. The architecture of the OPOSSUM DTSM.

5.2 System Architecture and Implementation

Figure 5.1 shows the overall architecture of OPOSSUM, which consists of three main components: the *schema manager,* which permits users to view, create, and edit schemas, the *meta-creator,* which allows new models and metaphors to be created, and the *storage manager*, which interfaces with the file system and DBMSs. These components are described in further detail in the following subsections. It should be noted that the components of OPOSSUM are generic, able to work with any models and metaphors defined in the formalism of Chapter 4.

5.2.1 The Schema Manager

The schema manager supports schema creation/modification and exploration and can be used by experts and non-experts alike. It consists of two modules. Module I manages models, metaphors, and schemas. It receives descriptions of data models, visual models, and metaphors, and then handle schemas of these models. Just as a database system can manage data given a schema, Module I can manage schemas given this model and metaphor information. At run-time, models, metaphors and schemas are represented as instances of C++ classes, and all operations on them are managed by methods. Module II interfaces these capabilities with the user by way of InterViews [LCV88], which converts visual schemas to manipulable screen objects. Given any specific visual model and metaphor, this code provides direct manipulation tools for creating, modifying, and exploring schemas, and menus for invoking commands that affect the models and schemas.

5.2.2 The Meta-Creator

The second component, the meta-creator, supports creation/modification of models and metaphors and is intended to be used by relatively knowledgeable people. Models and metaphors are specified as expressions in a formal grammar which is parsed into instances of the C++ classes used by the schema manager. These expressions resemble structure definitions in the C language, and should be straightforward for a knowledgeable person to understand and change. An example of this grammar appears in Figure 5.2, which shows a definition of the MOOSE data model. A description of the Meta-Creator language and a more detailed example may be found in Appendix A.

```
/* attribute type definitions */

AttrTypeDef string CharStringAttrType;
AttrTypeDef Class_Kind_Type CharStringAttrType
  Values {"tuple", "set", "multiset", "sequenced-set", "indexed-set",
          "primitive"};
AttrTypeDef DefaultToBlank CharStringAttrType;
AttrTypeDef ReverseCardinalityType CharStringAttrType
   Values {"1:", "M:"};
AttrTypeDef CardinalityType CharStringAttrType Values {":1",":N"};
AttrTypeDef RelKindType CharStringAttrType
   Values {"Has-Part","Is-A", "Collection-Of", "Associated-With",
           "Collection-Indexed-By"};
AttrTypeDef bool BoolAttrType;
AttrTypeDef ClassType PrimValuedAttrType Types {Class};

/* definitions for concepts */

PrimTypeDef Class{
  AttrDef ClassName string;
  AttrDef ClassKind Class_Kind_Type;
};

PrimTypeDef Relationship{
  AttrDef RevLabel DefaultToBlank;
  AttrDef ForLabel DefaultToBlank;
  AttrDef RevCardinality ReverseCardinalityType;
  AttrDef ForCardinality CardinalityType;
  AttrDef RelationshipKind RelKindType;
  AttrDef ForwardMutable bool;
  AttrDef BackwardMutable bool;
  AttrDef ForwardNull bool;
  AttrDef BackwardNull bool;
  AttrDef SourceClass ClassType;
  AttrDef DestClass ClassType;
};
```

Figure 5.2. The MOOSE data model an expression in the meta-creator formal grammar.

Ideally, the schema manager itself should be used for meta-creation. It is possible to define models and metaphors for visually representing models and metaphors, and such meta-models might make it easier for novice users to engage in meta-creation using the power of the schema manager. Meta-models have been defined for testing purposes, expressing the structure of models and metaphors as a directed graph, but so far the success of this approach has been limited because visual meta-schemas describing models and metaphors are very large, with several thousand nodes in the graph. As a result, meta-creation using the schema manager is a prime candidate for the use of abstraction, described in the next chapter. It remains to be seen how visual meta-creating will compare to that using the above grammar.

5.2.3 The Storage Manager

The third component, the storage manager, provides input/output of models, metaphors, and schemas to the first two components. Through generic code, it offers storage to ASCII files on disk. In addition, it can communicate schema information with external programs for further processing. When a model is defined for OPOSSUM, the definition includes a list of external programs that may be invoked by the user to process schemas of that model. Examples of such processing include sending the schema to the database, applying an automatic layout program to a visual schema, or producing statistics about a schema. These external programs are necessary because schemas in different models must be treated differently, e.g., a relational schema will be sent to a different DBMS than an object-oriented schema. The interaction between the storage manager and the external program works as follows: the model description of the external program includes a description of whether the external program is input-only, output-only, or input/output. If input is involved, the storage manager sends the current schema in a canonical, textual form to the program. The program executes, and if it does output, it creates a list of updates to the schema in the same textual form. The storage manager reads this list of updates and applies them to the schema.

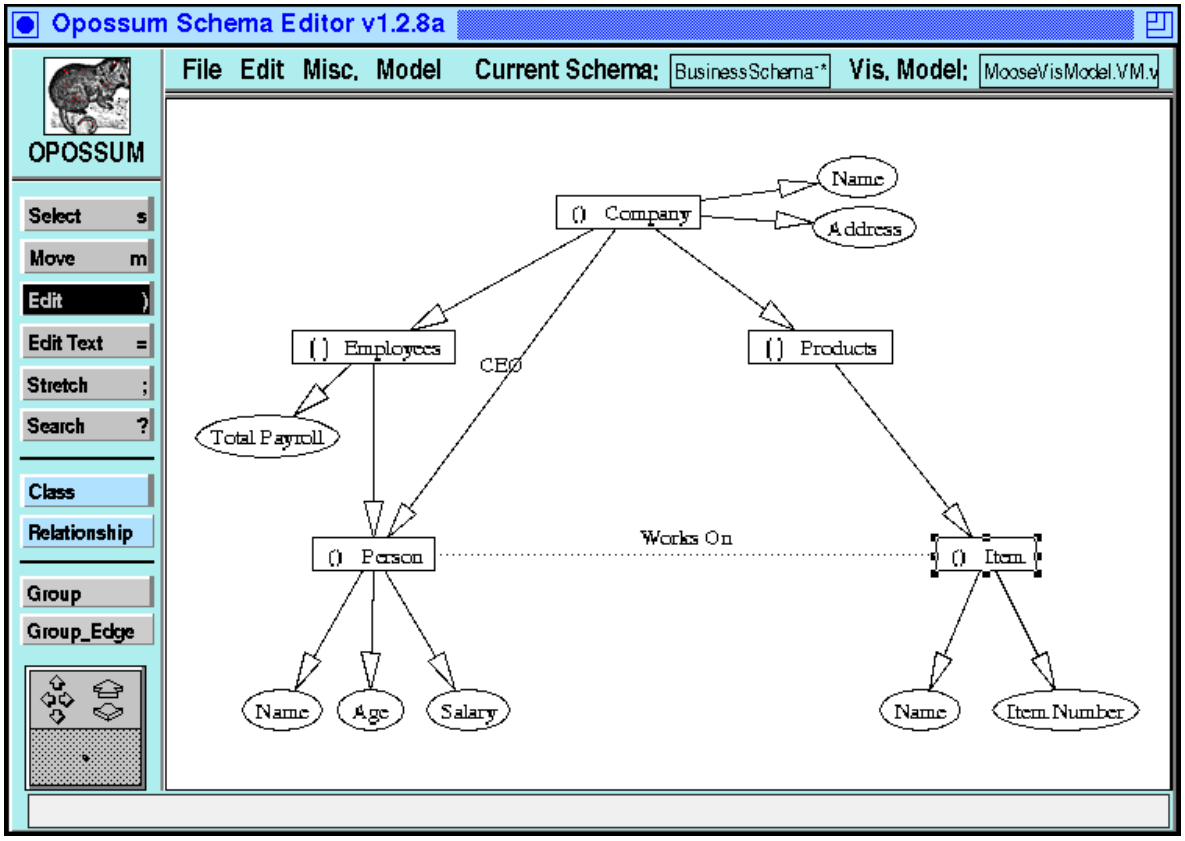
We should also mention that the Storage Manager separates the parts of a schema that are meaningful to the data model from those that are not. As a result, it is possible to store and retrieve many visual schemas in many visual models (or personal models) based on the same underlying data schema.

5.3 A ‘Demo’ of OPOSSUM

In this subsection, we describe how OPOSSUM operates and how users interact with it. Assume, for example, that a user wants to use OPOSSUM to manage MOOSE database schemas and prefers to view them as graphs. The user (or possibly a more knowledgeable designer) must first use the meta-creator to define the MOOSE data model, a directed graph visual model (like that seen in Figure 3.1), a metaphor between the two, and any external functions needed to communicate with the database. Once these models and metaphor are in place, the user can invoke the schema manager. Using the model descriptions, the system customizes itself to work with schemas of those models. An example of this customization is shown in the screen dump of Figure 5.2.

Near the top of the window, the *File* and *Edit* menus provide several generic capabilities for file operations (e.g., printing a schema or saving it to a file), as well as visual manipulation (e.g., cut, paste, zooming, panning, etc.). The *Misc* menu contains an option for each of the model-defined external schema processing programs mentioned in previous Section (e.g., it might include a program for sending the schema to a database). The *Model* menu provides access to the Meta-Creator. On the left, there is a column of buttons, each button defining a mode for direct manipulation of screen objects. There are generic buttons (e.g., to select, move, or stretch) and buttons specific to the visual model. In particular, there is a button for each concept of the visual model which, when selected, permits the generation of corresponding concept instances. When these types are mapped by the metaphor to concepts in the Data Model, the buttons are labeled with the Data Model type names. In Figure 5.2, there are four model-specific buttons, two for the data model (one for the Class concept and one for the Relationship concept), and two for the personal model (Group and Group_Edge, discussed in detail in the next chapter). By selecting the Class button, every subsequent mouse click on the drawing area generates a graph Node, which is mapped by the metaphor to a Class in the underlying data schema. By selecting the Relationship button, a subsequent mouse click starts the process for the generation of an Edge, in which the system asks the user to specify the two Nodes connected by the Edge.

The difference in system behavior for Node or Edge creation is not achieved by any specialized code written explicitly for the graph/MOOSE models. It is generic code that takes into account information that exists in the models. For example, the model specifies that the Edge concept has two attributes of type Node whose values may not be null. As a result, the system knows that it must ask the user to indicate two Nodes whenever an Edge is created. Thus, OPOSSUM uses the declarative definition of models and metaphors to capture not only appearance but behavior of visualizations as well.


Opossum Schema Editor v1.2.8a

File Edit Misc. Model Current Schema: BusinessSchema* Vis. Model: MooseVisModel.VM.v

Figure 5.3. A sample MOOSE schema created with the OPOSSUM tool.

In addition to creating instances of the various concepts of the visual model, a user may also want to change the values of the attributes of some of these instances. Spatial (location) and textual attributes can be modified directly through the *Edit-Text*, *Move*, and *Stretch* buttons. All other attributes are modified using the *Edit* button. When *Edit* is selected, a mouse click on a visual concept instance results in the appearance of a pop-up menu with all the concept's attributes that are relevant to the area of the click. By choosing one of the menu entries, a dialog box appears through which one may specify a value for the corresponding attribute. An example of the process is shown in Figure 5.3, where the system is in *Edit* mode, the menu with attributes of a Node concept has appeared (the Node being barely visible below the menu), and the text of the central label of the Node has been chosen for modification. Note that the attributes are distinguished into those that are mapped to the underlying data model (e.g., Label.text mapped to ClassName) and those that simply affect aesthetics. If a personal model had been specified as well, then the attributes in the pop-up menu would have been partitioned into three categories.

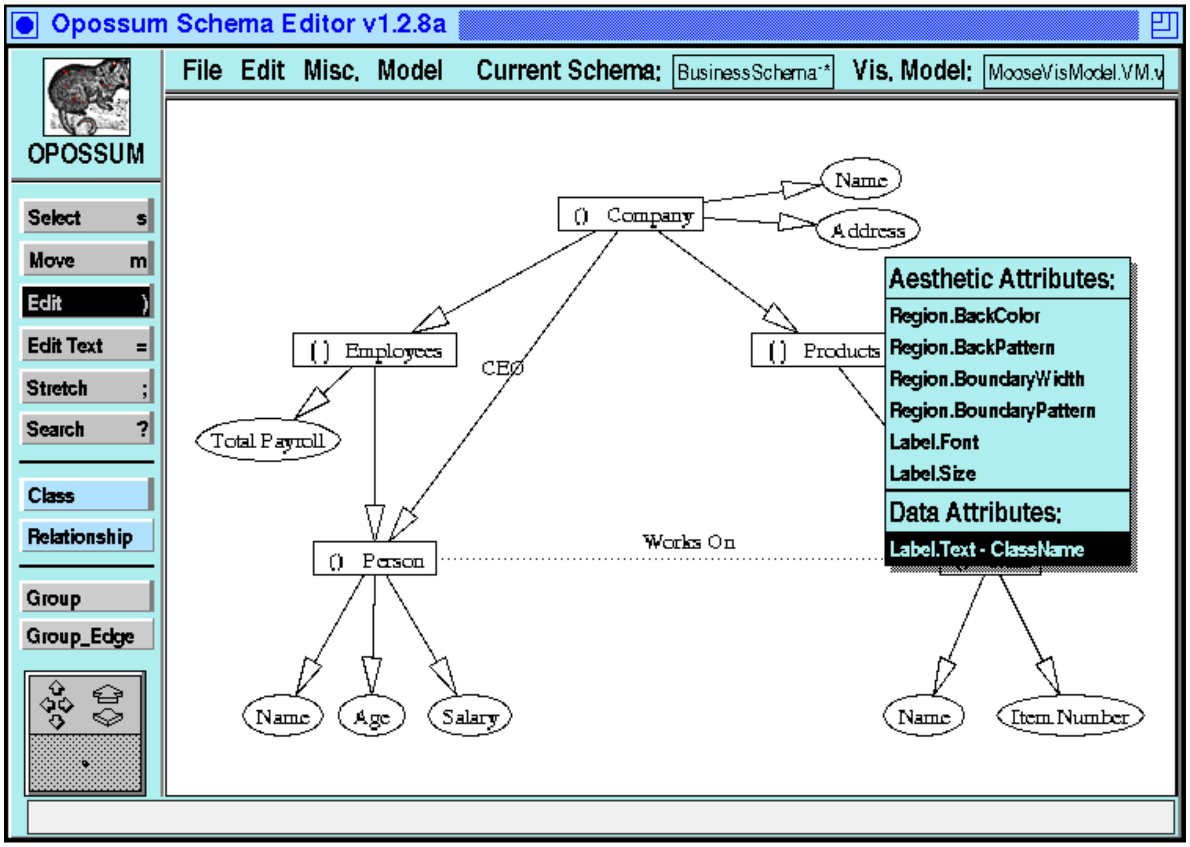


Figure 5.4. Changing attribute values in visual schemas.

5.4 Implementation of OPOSSUM

5.4.1 Implementation Status

The current implementation of OPOSSUM consists of about 31K lines of C++. It provides the entire spectrum of functionality described above, with the exception of a few issues on which work is still in progress (but close to completion). Specifically, the schema manager cannot yet store personal and aesthetic information in the underlying database; this is currently stored in ASCII files only. To support database storage of this information, it will be necessary to enhance the database with schemas for storing the personal and visual information, and to change the external program that ships schemas to the database. Another issue requiring more work is that the meta-creator does not use the capabilities of the schema manager, instead requiring models and metaphors to be specified as expressions in a formal grammar. Work is under way to allow meta-editing of models; a complete meta-model has been defined and is undergoing testing. Finally, the overlap-layout management algorithms described in Section 4.7 has been implemented only in a hard-coded manner for a single visual model. The architecture has been designed for general code to work with any model, but it is not yet complete.

Due to the diverse hardware environments of our collaborating scientists in ZOO, we have strived to maintain portability in the developed code. Thus, OPOSSUM currently runs on several Unix workstation platforms, including DecStations, HP Snakes, Sun SparcStations, and SGI Indigos. As for underlying database systems, we have developed code to communicate schemas to both Informix and the MOOSE-based database system under development for ZOO, which uses Informix as the underlying storage repository as well.

The next sections describe the C++ classes and algorithms used to implement the Schema Manager, Storage Manager, and Meta-Creator.

5.4.2 Schema Manager Module II - Editor Data Structures

All interaction between OPOSSUM and the user takes place using classes derived from InterViews [LCV88] and its graphical editor framework Unidraw [VL89]. The structure of these classes is shown in the upper box of figure 5.5. The central class is the ZooViewEditor class, each instance of which represents a window on the screen. Different subclasses of ZooViewEditor are used for the tools that edit schemas and queries and objects. Each ZooViewEditor instance is associated with visual and data Models, a Metaphor, a Schema, ZooGraphicComps, which maintains a list of VisPrims, each VisPrim associated with a concept instance. Several Editors can share the same ZooGraphicComps. Each ZooViewEditor instance also has one or more Viewers, which show one view of the information in in ZooGraphicComps. Each Viewer maintains a list of VisPrimViews, which represent VisPrims in a particular window.

Each subclass of ZooViewEditor is associated with a number of other classes that support interaction with the user: Commands, Tools, and Menus. Subclasses of Command are used for performing operations on VisPrims. Commands may be invoked either through selection of an item from a Menu, or as the result of the operation of a Tool. Tools are used for providing direct manipulation with VisPrimViews (such as Select, Move, Stretch, or Edit).

The VisPrim is a purely visual class; it captures information in terms of the four visual constructs described in Chapter 4: regions, text-displays, lines, and bit-maps. Each VisPrim is associated with an underlying VisualPrimitiveInst (described in the next subsection), which is part of the VisualSchema to which the Editor refers. All editing changes are made to the VisualPrimitiveInst, and propagated to the VisPrim.

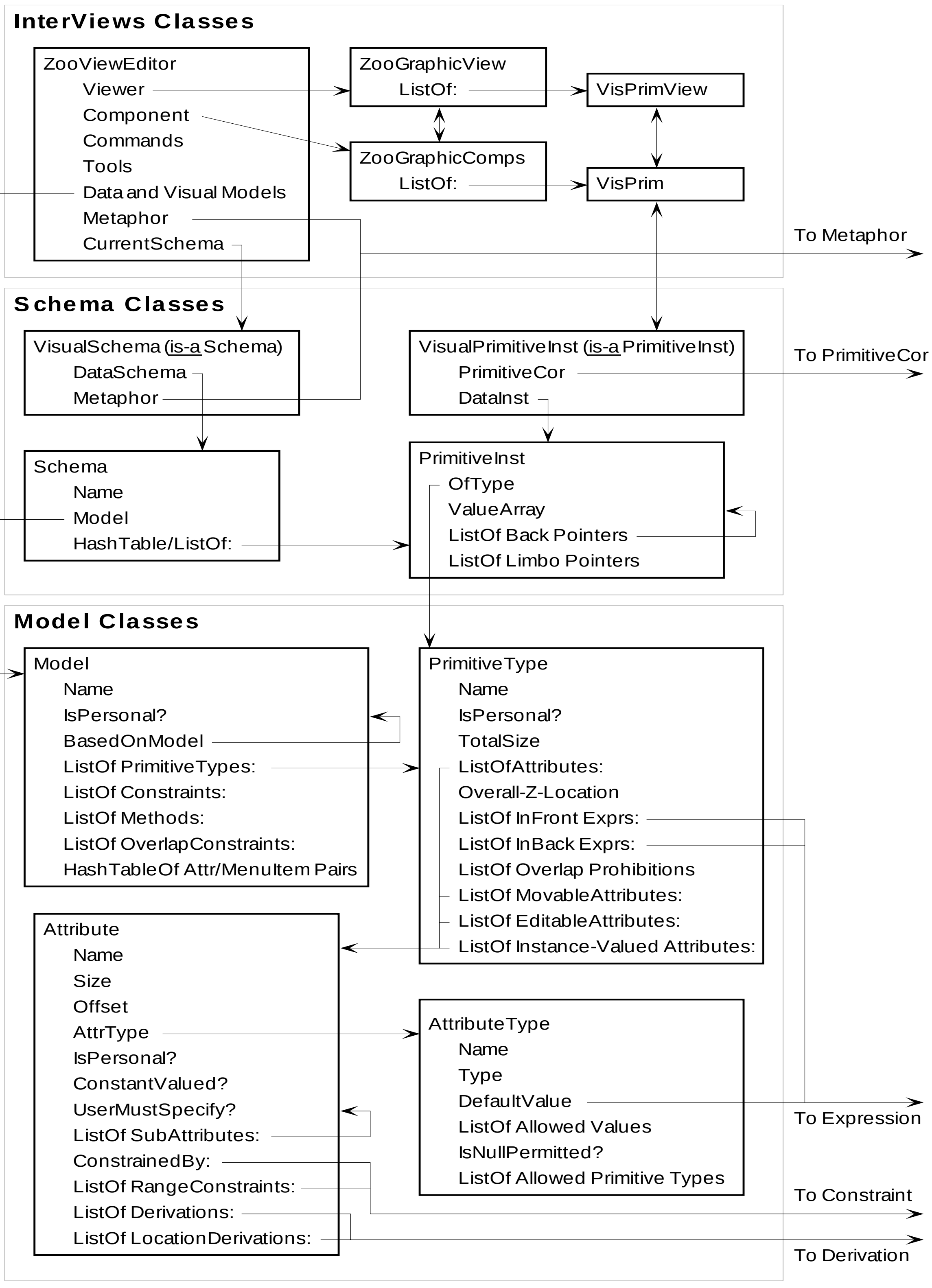


Figure 5.5. InterViews, Schema, and Model Classes

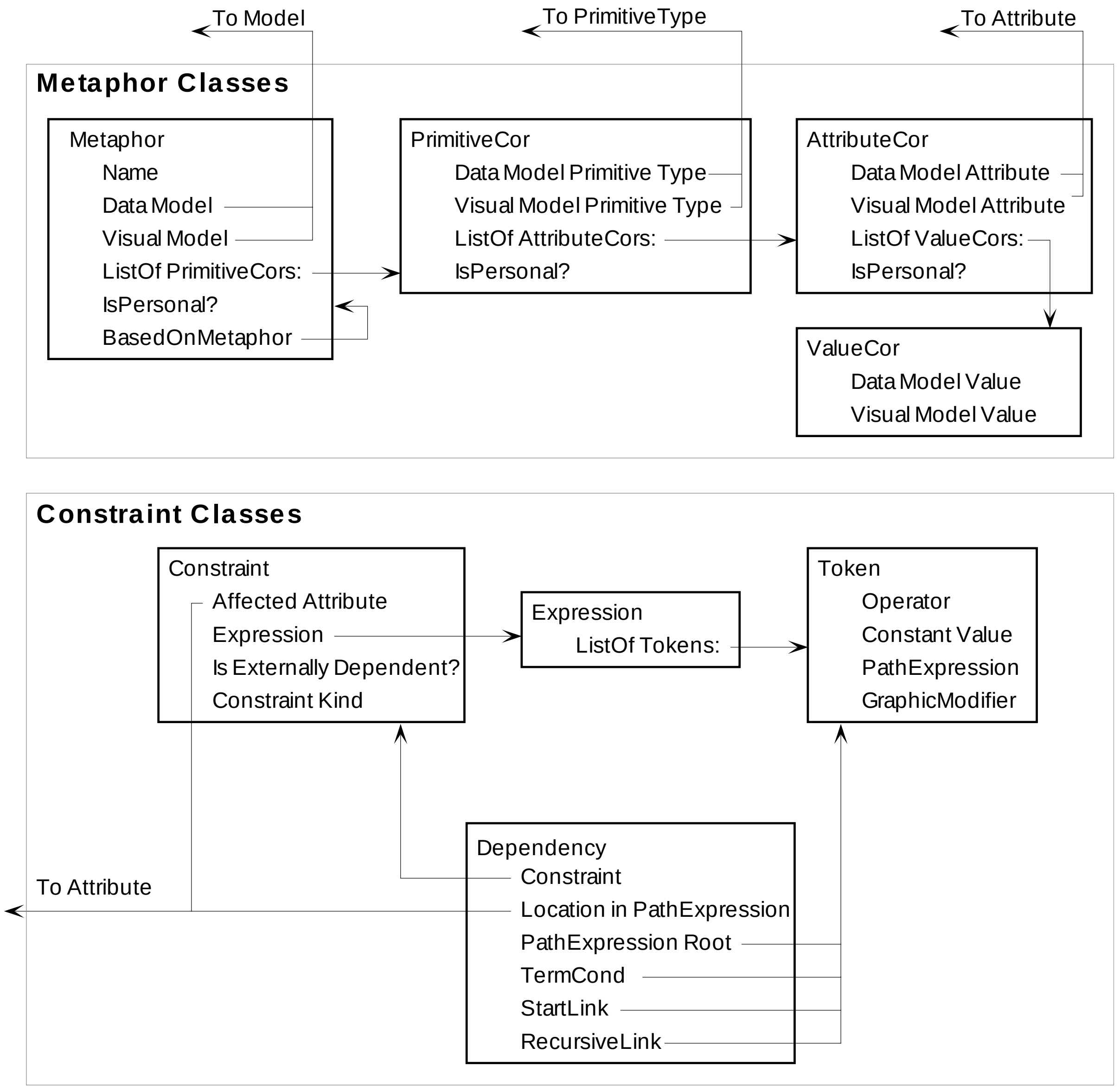


Figure 5.6. Metaphor and Constraint Classes.

5.4.3 Model Data Structures

Module I of the Schema Manager includes C++ classes for describing Models, Metaphors, and Schemas. The most important of these classes are shown in Figure 5.5. Models are described by a Name, a list of PrimitiveTypes (i.e., concepts) in the model, a list of external methods that can be invoked, and lists of constraints and overlap restrictions in the model. Models also include a hash table linking Attributes with MenuItems, so that MenuItems for the Edit Tool need not be created with each Edit operation. In addition, Models have a boolean value indicating if the model is a Personal Model, and if so a pointer to the data model of which the model is a superset.

The PrimitiveType class captures the concepts of a model through a Name, a list of Attributes, and a boolean value indicating if it is a personal concept. For visual types, the class includes an overall Z location, and two expressions used to determine where to place instances of the type with respect to being in front of or behind other instances. To manage overlap and layout, the type also includes a list of overlap prohibitions, each indicating a possible representation of this type, and all other types/representations with which overlap is forbidden. For efficient operation at run-time, the type maintains precomputed lists of Attributes that are movable and editable by the user, and also all those Attributes that are instance-valued.

The Attribute class captures concept attributes through a Name, an AttributeType, an indication if the Attribute is part of a Personal Model, if the Attribute's value may be changed by the user, and if the user must specify a value for the Attribute when an instance is created. Attributes may have a list of subattributes; the many Attributes of visual constructs such as regions or text-displays are associated with each other by being subattributes of the same parent. Constraints may be associated with an Attribute to determine or restrict its value. Given these constraints, the values of some attributes are dependent on others, thus dependency paths are precomputed so that when an Attribute's value changes for some instance, it is possible to trace back to all the Attributes of all the instances whose values must be recomputed. Location dependencies are a special case where the Attributes of one concept depend on the location of another, instead of a direct link through an instance-valued attribute.

The AttributeType class describes the type of an Attribute using a Name, a Type (indicating integer, floating point, character string, instance-valued, etc.), a list of allowed values if the AttributeType is enumerated, and an Expression for determining a default value for Attributes of the type. If the AttributeType is instance-valued, there is also a list of ConceptTypes whose value it can have, and an indication of whether null values are permitted.

5.4.4 Metaphor Data Structures

The visual metaphor as described in Chapter 4 is a mapping from a visual model to a data model. This is captured by the C++ classes described in the upper box of Figure 5.6. The Metaphor class has a Name, pointers to the visual and data Models, and a list of correspondences between concepts of the models. A Metaphor also includes an indication if it is mapping to a Personal data model, and if so, the Metaphor on which it is based.

Correspondences between concepts are described by the PrimitiveCor class, which includes pointers to the visual and data PrimitiveTypes, a list of correspondences between Attributes, and an indication if this correspondence is part of the Personal metaphor. The AttributeCor is essentially the same, except that it works on the level of Attributes. Finally, a ValueCor indicates a correspondence between two values as part of the metaphor mapping.

5.4.5 Schema Data Structures

The four classes used for representing schemas are shown in the middle box of Figure 5.5. The classes, VisualSchema and VisualPrimitiveInst are subclasses of Schema and PrimitiveInst respectively. The Schema class describes a schema in terms of a name, a Model, and a list of PrimitiveInsts. The PrimitiveInst includes a pointer to a PrimitiveType, and an array of values, one for each attribute of the PrimitiveType. For efficient operation at run-time, the PrimitiveInst maintains a list of backpointers than includes all other PrimitiveInsts who have an instance-valued Attributes pointing to this PrimitiveInst. In order to support undo of deletion operations, after deletion a PrimitiveInst can record the values of its instance-valued Attributes in the "LimboList." This way a PrimitiveInst may be deleted, and be removed from other instances lists of backpointers, yet still be able to restore all its Attribute values if the deletion is undone.

The VisualSchema class contains pointers to a Metaphor and the data model Schema from which it is derived, in addition to the information of a Schema. The VisualPrimitiveInst class is similar, it additionally contains pointers to the data model PrimitiveInst with which it is associated, and the PrimitiveCor used to map between the two.

5.4.6 Constraint Data Structures and Evaluation

The current implementation of constraints permits the value of one Attribute to be determined with respect to other Attributes (or constant values) in what is, in effect, a derivation. It is also possible to specify constraints that limit the range of values allowed for an Attribute, and constraints that cause the value of one Attribute to change in lock-step with another. The Constraint class is used to capture all of these in terms of the affected Attribute, an Expression, and an indication of the kind of constraint. The Expression class is a list of Tokens, and can be evaluated to determine the value of a derivation, whether or not a value is acceptable to a range constraint, or which attribute to change for a lock-step constraint.

The Token class describes one element of an Expression, which is either an operator (such as “plus”, “equal-to”, “and”, “max”, etc.),[8] a constant value, or a path expression (which is a list of Attributes). Sometimes there are visual values not directly described by Attributes. For example, the text-display visual construct includes Attributes for the font, size, text value, and center location of a piece of text on the screen. There is no Attribute for directly describing the location of the right-hand side of the text-display, though that information is implied by the values of the other Attributes. To express such values, Tokens include the GraphicModifier field, which can indicate if the left, right, top, or bottom of a visual construct is required. Thus the GraphicModifier changes the meaning of a path expression to indicate some visual aspect of the path expression’s terminating attribute.

When a model is first created, the set of constraint dependencies is computed. For each Attribute whose value is determined by a Constraint, an instance of the Dependency class is created for every other Attribute on which the first Attribute depends. These dependency records are stored with the other Attributes, so that when the value of one of those Attributes changes, all dependent Attributes can recompute their values. The Dependency class maintains pointers into existing Constraints, describing where in a Constraint Path Expression a particular Attribute is used. When the value is changed of an Attribute for a particular instance, the PathExpression from the Dependency is traced backward through the schema graph to determine what instances are affected.

For example, in a graph visual model, the location of the endpoints of an Edge are determined by the X and Y locations of the region of the Nodes it connects. Thus, there will be a Derivation in the DerivationList of the Node’s Region.X and Region.Y Attributes, and also the Edge’s From-Node and To-Node Attributes. Whenever a Node instance changes location, the path from Edge to Node is traced backward from the Node instance to see if any Edge instances point to that Node instance. If so, the Edge instances’ end or start locations are recomputed using the Constraint. Similarly, if the From-Node Attribute is changed for an Edge instance, the path is traced trivially to that same Edge instance, and its start location is recomputed.

5.4.7 Implementing Semi-Automated Layout

At the present time, the approach to semi-automated layout described in Section 4.7 has been implemented only in a hard-coded fashion for two specific models. A more general implementation that will work with any model is in progress, and will be complete soon. In the current implementation, safe locations are stored as

[8]The complete list of operators may be found in Appendix A in the description of the “expr” non-terminal of the Meta-Creator grammar.

additional integer-valued Attributes of the PrimitiveTypes that may not overlap. Extra code updates the safe locations, and checks for overlap after each change to the schema. Overlap is detected by a linear search through the instances in the schema, with all instances considered to occupy the footprint of their bounding rectangle (this makes computing overlap simpler). When an instance must be moved towards its larger safe location, it is moved incrementally, each step sufficient to either get it to its safe location, or else eliminate the current overlap. When an instance is moved towards its smaller safe location, the distance it can move is computed by finding all instances within the parallelogram between its current location and its small-safe location, and only moving as far as the closest one.

5.5 Summary

In the natural world, the Opossum is an animal known for its acting ability; the expression “playing ’possum” originates from its inclination to feign death when startled or alarmed. The OPOSSUM system is also an actor: given the script of model and metaphor definitions, it can assume the role of a direct manipulation editor for schemas from virtually any data model visualized in diverse ways. Not only can OPOSSUM play many different roles, it can also combine them into a single, mixed role. In the next chapter we describe some of the roles that OPOSSUM has played.