FROG and TURTLE: Visual Bridges Between Files and Object-Oriented Data[*] [†]

Vaishnavi Anjur **Yannis E. Ioannidis**[‡] **Miron Livny**
Department of Computer Sciences, University of Wisconsin, Madison, WI 53706
{vaish,yannis,miron}@cs.wisc.edu

Abstract

The problem of translating database objects into a flat format to be written out in a flat Ascii file or, conversely, translating the contents of a file into a complex database object arises in several applications. It is especially important in scientific database applications, where file-based communication with external programs (e.g., visualization packages or model simulations) is very common. We introduce *Frog*, a visual tool that can be used to specify translations between database objects and flat files, requiring no programming by the user. The tool can deal with objects of arbitrary complexity, without the object complexity being directly reflected in the complexity of the corresponding visual interaction. Based on the visual actions of the user, the tool stores enough information in a *map-file*, whose contents are used at runtime by another tool, Turtle, to translate any chosen database object into the appropriate file layout. The tool has been developed as part of the ZOO desktop Experiment Management Environment and has been used by a few experimental scientists with success.

1 Introduction

Many applications exist that require an Object-Oriented Database System (OODBMS) to be communicating with other, external, software, sending data to it or receiving data from it. For example, database data may be sent out for some complex computations to be performed on it (possibly by legacy software) with the results coming back to be stored in the database as well. Very often this communication is achieved through Ascii files (or Ascii byte streams) because of the heterogeneity of the overall system. For example, the external software may have been built independently of the OODBMS and may support a file-based I/O interface; or it may simply support an interface that is different from that of the OODBMS and files are a convenient intermediate representation for the data to be communicated. Hence, the OODBMS must have the ability to translate any complex database object into a flat structure to be written out into a file, or to read in a file and translate its flat contents into a complex object. This ability is also necessary in order for legacy data to be loaded into a database so that future access to the data can be done through the OODBMS.

The above functionality is especially critical for scientific databases, since they are associated with applications that have all the above characteristics. For example, data stored in scientific databases often needs to be sent out to specialized visualization programs to be visualized, or to statistical software for analysis, or to modeling packages for simulation, with the results (of the last two types of software) stored back into the database. Experiment management, which is our emphasis and main motivation, also generates similar scenarios. An experiment management system needs to communicate with the experimentation environment (be it a model simulation program, an automated assembly of instruments, or even a human that "manually" operates on instruments) to send requests for experiments with specific input and to later receive the experiment output for storage.

Translation between database data and files is often hard-coded into existing systems, e.g., bulk loading facilities in most commercial OO and relational DBMSs. This is undesirable, however, because it restricts the layout of the files that can be generated or loaded, while in scientific database applications, one wants genericity so that communication with several diverse programs is possible. An alternative is for the OODBMS to provide hooks so that different translation modules may be built, as application code. Such coding, however, could be rather laborious since

- it has to be repeated for each external software program expecting or producing files of different layouts;
- it has to deal with database objects of nontrivial structural complexity, e.g., sets or indexed-sets, and to produce the correct layout of the complex objects' parts, e.g., using the correct delimiter between set elements or the correct set terminator.
- it has to deal with format conversions between database values and their corresponding entries in files, e.g., between floating point and decimal values.

A third alternative is for the OODBMS to provide a higher-level mechanism, e.g., a declarative language, for describing how specific translations from database objects to files must be done, e.g., as in relational report writers. Then, the OODBMS either generates translation code based on the description or interprets the description at run-time to perform the necessary translations.

In this paper, we follow the third alternative in its interpreted version. Our task is slightly more complex than that of report writers though. Whereas they are primarily used

[*] Work supported in part by the National Science Foundation under Grant IRI-9224741.

[†] We would like to acknowledge Tom Wang for his significant effort in implementing Frog and Turtle.

[‡] Additionally supported in part by the National Science Foundation under Grant IRI-9157368 (PYI Award) and by grants from DEC, IBM, HP, AT&T, Oracle, and Informix.

to *design* the layout of files (reports), we need to translate database objects into *given* file layouts, required by the external programs. We introduce *Frog*, a visual tool that we have developed and is used off-line to *map* (i.e., specify translations from) database objects to flat files. (For simplicity, our entire presentation focuses on translations from database objects to files only, but the reverse direction is handled following essentially identical principles.) It generates a file of its own, a *map-file*, which contains all the necessary information for object translations. At run-time, this file is consulted by another tool, *Turtle*, which interprets its contents to *translate* any chosen database object into the appropriate file layout. We present the visual mechanisms incorporated into Frog and the underlying techniques that make these mechanisms work. These are sufficiently general to allow translations of objects of arbitrarily complex structure, e.g., sets of sets, as well as translations of objects in different branches of isa hierarchies (with different structures). We describe the contents of the map-file that is generated by Frog and also how Turtle uses them for specific translations. This whole effort has been carried out within the context of the *ZOO* Desktop Experiment Management Environment that we have been developing [8]. Hence, we also discuss some preliminary experiences that we have had with the tools as they have been used by some experimental scientists in their database-to-files translations.

2 The ZOO Experiment Management Environment

Experimental studies in essentially every scientific discipline go through a similar life-cycle of several phases: design, experimentation, and data analysis. In most cases, the current state of the art forces scientists to use different tools in each phase of that cycle, making the whole process difficult to manage. We are involved in the development of the ZOO desktop Experiment Management Environment that aspires to bring state-of-the-art management tools to the desk of experimental scientists [8]. ZOO is an integrated software package that will enable such scientists to manage their experiments and associated data from their desk via a uniform interface. This work is done in collaboration with researchers throughout the University of Wisconsin - Madison campus, especially the Departments of Soil Sciences, Biochemistry, and Physics.

ZOO has the abilty to communicate with several external experimentation environments, e.g., laboratory equipment, simulation programs, statistical analysis tools, etc. Each environment operates on an appropriate input file and produces an output file. An input file is constructed by a ZOO module called Turtle, which receives the oid(s) of the object(s) that capture the environment's input as well as the name of a map-file that has been generated off-line by another module called Frog. Turtle interprets the map-file, uses the given object oid(s) to extract the needed data from the appropriate database under ZOO, and eventually constructs the file. In a similar fashion, when the external processing is over, the output file produced is translated into database objects.

Turtle and Frog are the focus of this paper, so they are described extensively in subsequent sections. Understanding their operation requires some familiarity with Moose and Fox [12], the data model and query language of the database server of ZOO, which are briefly described below. There are three *kinds* of object classes in Moose: *primitive*, *tuple*, and *collection*. The primitive classes are *integer*, *real*, *boolean*, and *character-string*. Objects in tuple classes consist of a prespecified number of other objects, called *parts*, identified by labeled relationships. Objects in collection classes consist of an arbitrary number of other objects, all from a single *elements* class. Collection classes are further subdivided into *set*, *multiset* (bag), *sequenced-set* (list or array), and *indexed-set* classes. An indexed-set is a generalization of a sequenced-set. Whereas the members of a sequenced-set are indexed by the set of consecutive integers $\{1, \ldots, n\}$, for some n, the elements of an indexed-set are indexed by (the elements of) an arbitrary collection object. This collection object is called the *keyset* for the indexed-set.

There are five *kinds* of binary object relationships in Moose. An arbitrary number of *has-part* relationships, each pointing to a single object, defines the structure of tuple classes. A single *set-of* relationship defines the structure of collection classes, except for indexed-set classes whose structure is defined by a single set-of and a number of *indexed-by* relationships equal to the dimensionality of the indexed-set. *Association* relationships do not define any structure but simply connect individual objects in two arbitrary classes. Finally, an *is-a* relationship between two classes identifies one of them as a *specialization* of the other and implies that every object in the subclass belongs in the superclass as well. A *path expression* in Fox may traverse any of these kinds of relationships. It starts at a *source* class, and its result is the set of objects that are (transitively) related to the objects in the source class via the relationships in the path expression.

Using graphs to represent Moose schemas is very intuitive. Each class is a node: oval for primitives (abbreviated as *i* for integers, *r* for reals, *b* for booleans, and *c* character strings) and rectangle for all others. Each relationship is an arc in the graph, with different brush patterns indicating different kinds. Each relationship has a label in each direction, which if unspecified, is equal to the name of the target class of the relationship in that direction.

3 An Example

Figure 1 shows a simple Moose schema that is used as an example throughout the rest of the paper. It represents a (simplified) soil-science study to determine the total *yield* and *quality* of a crop depending on the *Weather* and on how various types of *plants* are distributed in a large field divided

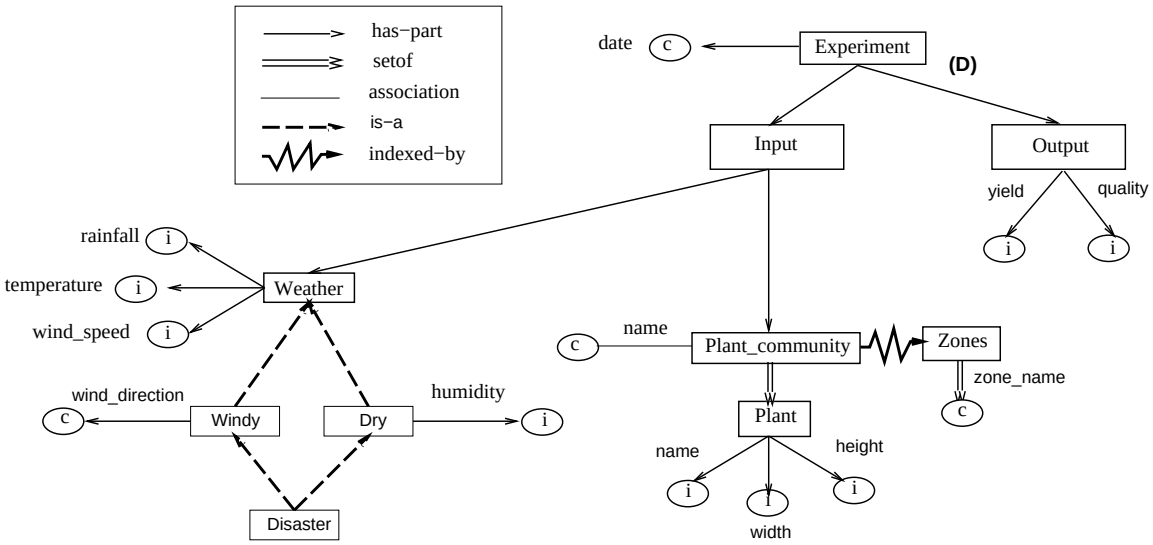

Figure 1: Sample Moose schema of Soil Sciences experiment

into *Zones*. Each experiment is modeled as a complex object, with sub-objects representing its *Input* and its *Output*. Its output is a pair of the total yield and quality of the harvest. Its input consists of the *Weather* and a *Plant_community*, which is an indexed-set of *plants* indexed by the set of field *Zones* so that the zone where each plant is grown is recorded. The weather is captured by *rainfall*, *temperature*, and *wind-speed* values, and may be *Windy* (wind-speed > 30 mph), in which case *wind-direction* becomes important as well, *Dry* (rainfall < 2 in), in which case air *humidity* becomes important as well, or *Disaster*, which combines the two.

We assume that the output part of an experiment is the result of running an external program on its input part. Figure 2 shows an example input file for that program, which contains the necessary information plus other items as well (the background patterns under some items will be explained shortly).

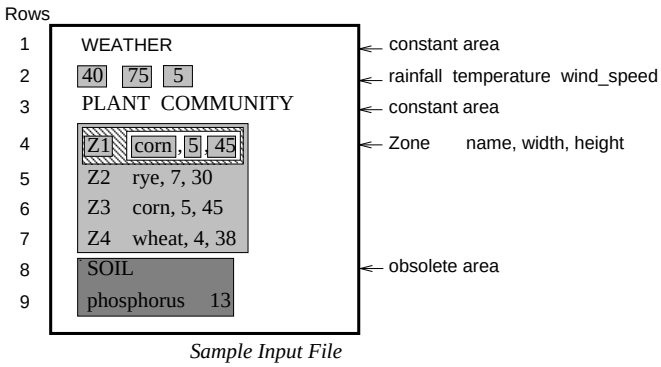

Figure 2: An example input file

4 General Operation of FROG

Before describing the basic operation of Frog, we first clarify some basic notions and introduce some terminology. Given a complex database object, one can write in a file only its parts, elements, or associated objects that belong to primitive classes, i.e., leaf classes in the schema graph. For example, one cannot write the oid of a plant object because it is meaningless outside the database system. Thus, when we use the terms 'mapping a complex class' (for Frog) or 'translating/printing a complex object' (for Turtle), we refer to the primitive, alphanumeric, objects related to it.

Also, in principle, there may be multiple independent classes in the schema whose objects correspond to the file to be generated. Nevertheless, for simplicity of presentation, we assume that there is only one such class. It is called the *source* class, and any object in it that Turtle must translate is called a *source* object. Generalizing to mapping multiple classes to a single file is straightforward.

Based on the above, we turn to a description of how Frog operates. The users of Frog are designers who are familiar with both the Moose database schema and the layout of the input file to the external program. A *sample* input file for the external program is brought on the main window of Frog. The sample file could exist from earlier uses of the external program or could have been constructed just so that it can be used in the mapping process. The Moose schema for the experiment concerned is also brought in graph form on another window managed by the Opossum schema manager [5]. The entire mapping task proceeds by repeating the following step sequence.

1. The designer chooses an *area* in the sample file (by highlighting it with the mouse). This signifies that mapping

specifications for that part of the file will be given. The area remains highlighted for the rest of the session with Frog and even across sessions. At this point, only areas that cover all the way to the end of a file line before moving on to the next line are supported.

2. The designer chooses a class in the schema shown by Opossum (by a mouse click). The path expression from the source class to the chosen class is sent to Frog. This signifies that the area identified on the sample input file in step 1 is mapped from the contents of the specified class.

3. Depending on the kind of class chosen in step 2, Frog presents another window with various entries constituting a *printing specification*, i.e., how an object of the class should be printed. Usually, only a subset of the entries is enough, but the system presents all those that could possibly be used. Based on the precise contents of the area chosen in step 1, some of the entries are already filled up by best guesses of Frog. The designer may accept those or modify them appropriately.

For example, assume that the file in Figure 2 is used as the sample input file for the mapping process. To map the rainfall attribute in the schema of Figure 1 to the area of the sample input file containing 40, the designer would first highlight 40 and then click on the oval pointed to by the arc labeled 'rainfall', which represents the class of integers, where rainfall values belong. Opossum generates the path expression Experiment.Input.Weather.rainfall and transmits it to Frog. The printing specification window then appears with entries related to integers, e.g., maximum length (number of digits) occupied by any integer (Section 5.1). This entry will contain the value 2 as a guess, which is the length of 40. If the external program does not need rainfall values printed so that they occupy a prespecified maximum length, or if the database may contain rainfall values that are longer, then the user will change the estimated value appropriately (in the first case, it will make the entry empty).

The step sequence outlined above deals with the areas of the sample input file that are indeed mapped to from data in the database, called *variable* areas. There could be areas, however, that should appear in *no* input file generated by Turtle. These may be found in *legacy* sample files that no longer correspond exactly to the input required by the external program. Such areas are called *obsolete*. They are identified as in step 1 above with Frog being in a special mode that interprets step 1 differently: the area is highlighted in different color from that of variable areas, and there is no continuation into steps 2 and 3. In Figure 2, the areas in rows 8 and 9 are obsolete, as indicated by the darker background.

In addition to the above, there could also be areas in the sample input file that should have the same value in *all* input files generated by Turtle and do not correspond to any class in the schema. These are called *constant* areas. Any area not highlighted through step 1 is considered constant. In Figure 2, the areas in rows 1 and 3 are constant, as they have not been highlighted at all.

5 Basic Translation Specification by FROG

For clarity of presentation, we first deal with mapping a restricted form of Moose schemas to files, and then remove the restriction in Section 6. In particular, we consider mapping (sub-)schemas that contain only tuple and primitive classes, and only has-part and association relationships. That is, when applied on a single source object, any path expression to a class that must be mapped results in a single object as well. In this case, all (printable) information about the source object is in its leaves. Hence, for the purposes of mapping to a file, the entire schema of interest can (although does not have to) be seen as equivalent to a simple 1-level-deep schema with all leaves hanging directly under the (tuple) source class.

5.1 Visual Language

In the restricted case above, the main concern is mapping the leaves of interest to the file in the appropriate order. The source class maps to the entire sample file, and its leaves in the schema to the individual elements in the sample file.

To express the above, designers interact with Frog as follows. Bringing the sample input file on the screen is equivalent to choosing the entire file area in step 1 of the step sequence of Section 4. The class indicated in step 2 becomes the source class of all path expressions to be generated by Opossum later on. Given the restrictions of this section, this must be a tuple class, so in step 3, the printing specification window for tuples pops up, containing the following entries:

- Delimiter between the parts of the tuple object
- Terminator for the tuple object
- Maximum number of elements per line
- Maximum number of characters per line
- Default value to be printed if database entry is null

Consider the example of Figures 1 and 2. If we only concentrate on row 2 of the sample file and the Weather part of the Input class in the schema, then we are dealing with an instance of the restricted case discussed in this section. By specifying the delimiter entry of the printing specification window to be `<space>` and the terminator entry to be `<return>` (the 'newline' character), and leaving the subsequent two entries unspecified, a designer provides enough information for Turtle to know how to print an entire tuple object in a file (all parts printed in the same line).

After this, designers enter the step sequence of Section 4 as many times as there are variable areas in the sample file. Independent of which primitive class the corresponding leaf in the schema is, the printing specification window that pops up always contains the following entries:

- Length (in number of characters)

- Default value to be printed if database entry is null

If the corresponding leaf in the schema is the class of reals, then the printing specification window also contains the following additional entries:

- Precision (in number of decimal digits)
- Notation (scientific or decimal)

Consider again row 2 of the sample file in Figure 2. The following table presents collectively printing specifications that designers may give as they map each of the three primitive parts of the Weather class:

Entry	rainfall	temperature	wind-speed
length	3	4	2
default	-1	-999	-1

As mentioned above, for each entry, Frog initially offers a guess for a value based on what the designer highlighted on the sample input file. (For lack of space, the details of this guessing as implemented in Frog are not presented here.)

5.2 Map-File

The result of a session with Frog is a map-file. For each variable and constant area identified through Frog, the map-file contains a *block* with all the necessary information to populate the area in an input file. There are different types of blocks for the different types of areas and/or kinds of mapped classes. Some blocks are *printable*, capturing areas of individual values that may be printed. Other blocks are *structural*, capturing larger areas that contain smaller (printable or structural) areas and enclosing the blocks corresponding to these contained areas. Each block begins with a "begin{X}" statement and ends with an "end{X}" statement, where X signals the type of the block, e.g., tuple, set, integer, string, constant. Between the two enclosing statements of blocks corresponding to constant areas, one finds a single line with the constant to be printed. For blocks of variable areas, one finds in separate lines the path expression corresponding to the block (step 2), all the entries constituting a printing specification for the type of the block (all those in the window of step 3), and possibly other blocks in the required order. For the restricted case of this section, the outermost block has X always equal to "tuple", and encloses inner blocks with X equal to "constant", "integer", "real", or "string", which enclose no further blocks.

Continuing on with the previous example, the map-file generated through Frog for a file that consists of only row 2 of Figure 2 is given below:

```
\begin{tuple}
pathexp = input
delimiter = <space>
terminator = <return>
maxelms =
maxchars =
default = NULL
  \begin{integer}
  pathexp = Input.Weather.rainfall
  length = 3
  default = -1
  \end{integer}
  \begin{integer}
  pathexp = Input.Weather.temperature
  length = 4
  default = -999
  \end{integer}
  \begin{integer}
  pathexp = Input.Weather.wind-speed
  length = 2
  default = -1
  \end{integer}
\end{tuple}
```

In the above, we assume that if the input oid that Turtle receives at run-time is null, then the string 'NULL' will be printed. Likewise, if any of the Weather parts is null, then the values -1, -999, and -1 are printed in place of rainfall amount, temperature, and wind-speed, respectively. Note how the information given as printing specification is repeated in the map file. Also note that, even if the designer specified mappings in an arbitrary order, in the end, Frog rearranges blocks to reflect the order of the corresponding areas in the sample input file. In reading the map-file, Turtle will generate an input file in the order it finds printable blocks, which should be identical to the spatial order of the corresponding areas in the sample input file, e.g., first rainfall, then temperature, then wind-speed.

6 General Translation Specification by FROG

In this section, we remove the restriction of Section 5. In particular, we describe how to deal with mapping set or indexed-set classes, and also how to deal with is-a relationships, which are features found in many applications. Even in the most general case, all (printable) information about the source class remains in its leaves, but now the intermediate structure is important for the overall layout of the file, capturing larger areas of it, and cannot be flattened out as in the restricted case of Section 5. This intermediate structure generates isomorphic hierarchies in the highlighted areas of the sample input file as well as in the corresponding map-file blocks.

Tuple classes are treated exactly as described in the previous section for the source class, except that step 1 corresponds to explicitly identifying a file area corresponding to a tuple object. Hence, we do not elaborate on them any further.

We discuss each of the remaining three advanced features in a separate subsection below. In all cases, the designer must follow the step sequence described earlier, with differences only in the printing specification window and the resulting map-file block. In general, the areas highlighted in step 1

may form a containment hierarchy in the sample input file. Each such area is highlighted in different color from its immediately enclosing area and is interpreted within the latter's context. Likewise, path expressions returned for an area at step 2 by Opossum are checked for having the correct prefix based on the path expression returned for the immediately enclosing area.

6.1 Sets

6.1.1 Visual Language

The printing specification window that pops up in step 3 of the sequence contains the following entries related to sets:

- Delimiter between the elements of the set object
- Number of elements in the set object
- Terminator for the set object
- Maximum number of elements per line
- Maximum number of characters per line
- Default value to be printed if set in database is null

Note that the only difference from the entries for tuple objects is an additional entry for the number of elements in the set object, something that is not needed for the parts of a tuple object as their number is prespecified in the schema.

After this, designers enter the step sequence again *once* to identify one element of the set. Depending on the type of the element (primitive, tuple, set, etc.), the sequence will proceed accordingly, based on our descriptions in the corresponding subsection. Note that this does not have to be repeated for every set element in the sample input file, as sets are uniform objects, and describing how to print one of them is enough.

The example of Figures 1 and 2 contains no set classes that are to be mapped on their own to the sample file. The only set class, Zones, is the keyset for Plant-community and must be mapped as part of the indexed-set. The whole process for sets, however, is extremely similar to that for indexed-sets, so the example that we give in Section 6.2 illustrate the situation for sets as well.

6.1.2 Map-File

The block inserted in a map-file for a set begins with a "begin{set}" statement and ends with an "end{set}" statement. Between these two statements, one finds in separate lines the path expression corresponding to the block, all the entries constituting a printing specification for sets, and a block for the elements.

6.2 Indexed-Sets

The following discussion deals with one-dimensional indexed-sets, i.e., indexed by a single collection. The generalization to multi-dimensional indexed-sets is straightforward, as one may view them recursively as one-dimensional indexed-sets of indexed-sets with one fewer dimension.

6.2.1 Visual Language

Again, the printing specification window that pops up in step 3 contains exactly the same entries as that of sets, except that now one deals with key-element pairs of the indexed-set instead of simply elements. (A key-element pair consists of a member of the keyset (key) and the corresponding member of the indexed-set (element).)

In Figure 1, Plant-community is an indexed-set class, indexed by the Zones set class. To specify the appropriate mapping, designers would first highlight the large light-gray area of the file, as shown in Figure 2, then click on the Plant-community node on the schema, and finally possibly update the entries in the printing specification window. In this case, except for the ever-present default value, all that is needed is to put `<return>` in the delimiter entry, signifying that each key-element pair occupies a single line.

After this, designers enter the step sequence again *once* to identify one key-element pair of the indexed-set. Due to the special nature of indexed-sets, a deviation from the norm is necessary here: step 2 is skipped as there is no schema class corresponding to key-element pairs. Since the area identified in step 1 is within the indexed-set area, however, Frog has the necessary information to skip step 2, and simply proceed to step 3 by presenting a printing specification window. This is identical to the window for a tuple, as the key and the element in the pair are essentially two individual entities.

After that, designers will have to enter the step sequence twice more, once for the key and once for the element. Each case will proceed normally, based on the types of the key and the element, respectively.

Continuing on with the example of Figures 1 and 2, the key-element pair is identified in the file by highlighting the first such pair, the line starting with Z1. In Figure 2, this is shown with angled striped background. Frog will then go directly to step 3 to allow the designer to specify how the pair will be printed. After that, the designer will enter the step sequence once by highlighting the key Z1 (light-gray area) and clicking on the character string class of members of Zones, and once by highlighting the corresponding element 'corn,5,45' (white area) and clicking on the plant class, which is the members of Plant-community class. Finally, the designer will enter the step sequence again, once for each part of the plant tuple class (small light-gray areas under corn, 5, and 45). The printing specifications appropriate for this file are captured in the generated map-file below, so they are not repeated here. Note that one needs to map to a single key-element pair of the indexed-set, as indexed-sets are homogeneous collections and each pair will be printed in the same way. Also note that the precise shade of the background that each area receives is not important, as long as it is different from the immediately enclosed and the immediately enclosing areas. Only obsolete areas have a standard shading to distinguish them from all variable areas, which in our example is dark gray.

If the indexed-set is of the sequenced-set kind (i.e., indexed by the set of integers), then Frog treats it as a regular set since for printing purposes there is no difference between the two. Thus, when an area is highlighted within an indexed-set area in step 1, Frog does not enter the special process of skipping step 2 so that a key-element pair is identified. Instead, it enters step 2 as usual, waiting for the element class of the sequenced-set to be identified through Opossum.

6.2.2 Map-File

The block inserted in a map-file for an indexed-set begins with a "begin{indexedset}" statement and ends with an "end{indexedset}" statement. Between these two statements one finds information identical to that of a set block. For a general indexed-set, the enclosed block for the elements is a tuple block, which in turns encloses two further blocks, one for the keys and one for the elements. For a sequenced-set, the enclosed block is of the type corresponding to the elements of the sequenced-set.

Continuing on with the previous example, the map-file block generated through Frog for the Plant-community indexed-set based on the sample file of Figure 2 is given below:

```
\begin{indexedset}
pathexp = Input.Plant-community
delimiter = <return>
noelms =
terminator =
maxelms =
maxchars =
default =
  \begin{tuple}
  pathexp =
  delimiter = <space>
  terminator =
  maxelms =
  maxchars =
  default =
    \begin{string}
    pathexp = Input.Zones.zone-names
    length = 2
    default = NULL
    \end{string}
    \begin{tuple}
    pathexp = Input.Plant-community.Plant
    delimiter = ,
    terminator =
    maxelms =
    maxchars =
    default = NULL
      \begin{string}
      pathexp = Input.Plant-com.Plant.name
      length =
      default = unknown
      \end{string}
      \begin{integer}
      pathexp = Input.Plant-com.Plant.width
      length =
      default = -1
      \end{integer}
      \begin{integer}
      pathexp = Input.Plant-com.Plant.height
      length =
      default = -1
      \end{integer}
    \end{tuple}
  \end{tuple}
\end{indexedset}
```

6.3 Is-A Hierarchies

Consider an object class that is the root of an is-a hierarchy. When an object in that class is translated into file format, the resulting file layout is different depending on the particular subclass where the object belongs. How all the possible layouts are captured in a single map-file is the topic of this subsection. The key problem is how to avoid remapping areas in the file that are common to objects of all subclasses. During the mapping process, designers may need to use several sample input files, as many as there are different file layouts in the worst case. In the following discussion, we assume that the same object can be in two classes only if one is a subclass of the other, or if they are both superclasses of a third class in which the object belongs as well (multiple inheritance).

6.3.1 Visual Language

Designers choose to start working on one of the sample input files. The step sequence for the root class of the is-a hierarchy proceeds on that file as described above based on the kinds of the classes mapped. Let the *root area* be the file area corresponding to the root class object in the file. When an area enclosed in the root area is highlighted in step 1 and then associated with a subclass of the root class in step 2, the comparison of the two path expressions reveals the relationship to Frog, which generates the appropriate special block in the map-file.[1] This may be repeated recursively for an entire is-a path from the root class down to increasingly specialized subclasses.

To capture a different file layout, corresponding to a different path in the is-a hierarchy, designers bring up another sample file. That second file could be a full-fledged input file, or simply a file that contains just the areas that are different from the original file, or anything in between. Step 1 of the first sequence with this file identifies the area that has different layout from the first file and needs mapping, while step 2 identifies the subclass that corresponds to this area. Beyond this point, the process continues as before.

The above process is most efficient when all the elements of a superclass are mapped in association to the superclass, and that mapping is shared with all the subclasses. However, the ability to do that depends primarily on what is common

[1] Full path expressions in Fox include explicitly traversals of is-a relationships as well, with @ being the connective indicating such a relationship.

among the various file layouts. If the common elements are placed close together and in the same layout for objects in the different subclasses, then they can be mapped only once. If they are interleaved with elements that differ from subclass to subclass, or they are laid out differently for objects of different subclasses, then the common part may become minimal or even nonexistent.

Finally, for multiple inheritance, mapping for the common subclass must repeat the mapping for the elements of all but one of its immediate superclasses.

As an example, we present a case that is relatively simple, which can be dealt with without using many of the features of the general algorithm described above and in fact requires a single sample input file. Consider again the schema in Figure 1, where the Weather class is the root of an is-a hierarchy that includes multiple inheritance. The original Figure 2 has a general Weather object, since it includes none of the specialized attributes. Figure 3 has a Disaster Weather object, since it includes all five of the possible Weather attributes.

Designers could use the general algorithm to map each subclass separately to an appropriate input file. However, in this case, all mappings may be specified by simply using the file in Figure 3, which contains a Disaster Weather object. Because the subschema that deals with Weather contains only a tuple class with has-part relationships, as in Section 5, mapping may proceed directly with the leaf classes. After mapping rainfall, temperature, and wind-speed, as before, the designer may enter the step sequence by highlighting the area that contains NW and then clicking on the character-string class connected to the Windy class. The path expression returned indicates that this is for a subclass of Weather, which provides enough information for Turtle to do the appropriate translation in each case. The designer may proceed similarly to do the mapping for humidity, by highlighting the area that contains 52 in the sample input file.

6.3.2 Map-File

The block inserted in a map-file for any subclass begins with a "begin{isa}" statement and ends with an "end{isa}" statement. It is always within the block of a superclass of it (most probably its immediate superclass), which may also be an "isa" block. The blocks of sibling subclasses (i.e., with a common immediate superclass) are enclosed in the same superclass block but not in each other.

Continuing on with the previous example, if we only concentrate on row 2 of the file in Figure 3, the map-file block generated through Frog for the entire Weather is-a hierarchy is the one shown in Section 5.2, but with the following additional blocks placed immediately before the final "end{tuple}".

```
\begin{string}
pathexp = Input.Weather@Windy.wind-direction
length = 2
default =
\end{string}
\begin{integer}
pathexp = Input.Weather@Dry.humidity
length = 3
default =
\end{integer}
```

Note that by having the default value for wind-direction and humidity be empty, if Turtle deals with a Windy object, the path expression leading to humidity will return a null object and therefore nothing will be printed, as it should. Similarly for the wind-direction attribute if Turtle deals with a Dry object. The layouts of the input files have permitted the designer here to work with a single sample input file and generate no isa blocks.

7 General Operation of TURTLE

When the external program needs to be executed with a specific input expressed as a Moose object, an input file with the appropriate layout has to be generated and filled with the right values from the database object. Turtle accomplishes this based on the map-file that has been generated for that purpose and the oid of the input object.

As mentioned earlier, the blocks in the map file are placed in the order their corresponding areas appear in the file, except for sibling is-a blocks, which appear in an arbitrary order between them within the same superclass block. For every block, Turtle sends to the database server a query that essentially processes the block's path expression, retrieving the object(s) that correspond to the file area related to the block, if any. If the path expression leads to a primitive class, the retrieved value(s) should actually be placed in the input file in that area. This placement is done by Turtle interpreting the contents of the printing specification windows of the area and all its enclosing areas, and deriving the appropriate printing format for the value(s).

When is-a blocks are nested one inside another, they are processed as all other blocks, the outer before the inner. When they are siblings, they are processed in the order they appear. In the absence of multiple inheritance, at most one of the sibling blocks may have its path expression return a nonempty answer, since objects cannot appear in multiple classes without a common subclass. In the presence of multiple inheritance, multiple sibling blocks may return nonempty answers, in which case Turtle must look further inside for the block corresponding to the common subclass to determine which of the sharing superclass blocks should be followed.

8 Status, Experience, and Other Uses

Preliminary versions of Frog and Turtle have been implemented in C++ in the context of the ZOO system. They do not deal with indexed-sets or is-a hierarchies but support all the remaining features described above. The missing items are currently under implementation. These early versions have

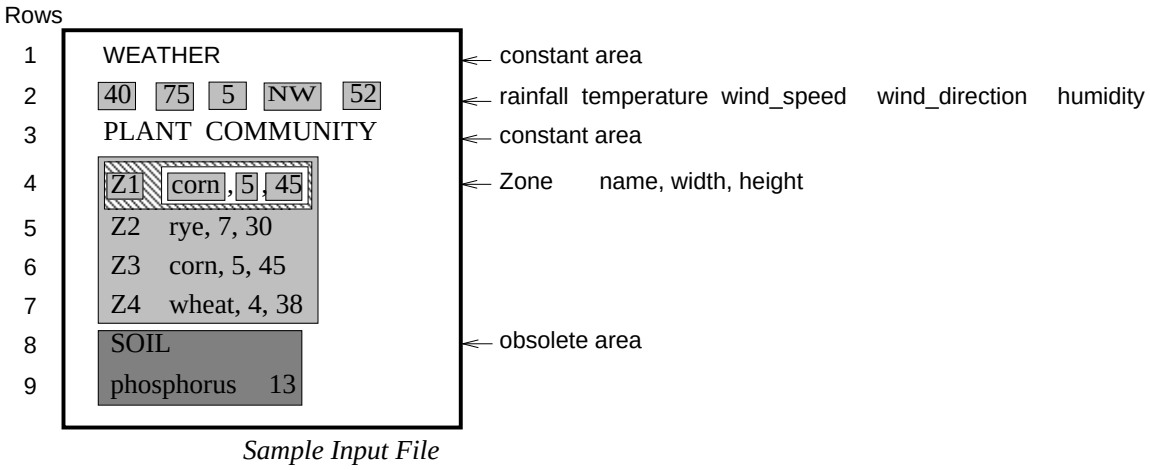

Figure 3: An example input file for Disaster Weather

been used by scientists in the Soil Sciences Department and the Biochemistry Department of the Univ. of Wisconsin, to automate generation of input files to their experiments. Their input files are very large (containing hundreds of parameters and many constants), and the use of tools like Frog and Turtle has significantly simplified the task of generating these files.

In addition to communicating with external programs, Frog and Turtle can also be used as reporting tools. For instance, users may like to print copies of details on each plant specimen they studied. A sample file with the desired layout is first created, Frog is then used to map the corresponding database classes to the file elements, and finally Turtle generate the required copies.

Frog and Turtle can also be used for data migration between heterogeneous DBMSs. Data from the source system can be first transferred in a file based on the layout expected by the target system, which can then read and translate the file into its internal representation.

9 Related Work

Most commercial DBMSs provide utilities (such as report writers) to transfer database data into files in user-specified layouts. There are also numerous tools that enable information exchange between different systems via files. Although often similar in functionality to Frog and Turtle, these tools usually work in restricted domains and do not emphasize visual interaction like Frog and Turtle.

We first mention such tools that have been developed in the context of scientific databases, as this has been our primary motivation as well. The Computational Chemistry Database project [3] provides inter-operability between computational chemistry applications by encapsulating the applications and data in an object-oriented database. The system is based on the framework developed by Cushing et al. [2] for interfacing experiments with a database and has been demonstrated to serve the needs of computational chemists. The system translates data between databases and files that have well defined layouts. The layout of the files is specified using *CCIL* and *CCOL* (Computational Chemistry Input/Output Language). These are declarative report-writing languages that are tailored towards the requirements of chemistry applications.

Chipperfield et al. address the issues of storing genome data into a relational database, migrating data from other sources into that database, and retrieving data from that database [1]. The database can be interfaced with applications that process the data in well-defined formats for the genome.

Both the above tools were developed to provide interoperability between specific applications, so they work primarily within the specific domains for which they were built. Our system, on the other hand, is general-purpose and hence well-suited to a broader set of applications. In addition, it offers a visual user interface that significantly affects usability.

Similar efforts exist outside the scientific database realm as well. In some approaches, the contents of an Ascii file are described by means of a grammar. The OBST persistent object management system [11] (developed as part of the STONE [9] environment) supports a tool called the Universal Structurer and Flattener (STF). If the structure of a file can be described by a context-free grammar, STF can generate a *parser* (structurer) and *unparser* (flattener) for the file. The structurer parses the contents of the file according to the grammar and creates an object structure and the flattener does the inverse operation. The grammar file resembles the map-file generated by Frog. The disadvantage of this approach is that the scientist has to understand the language in which grammars are specified and then design the grammar desired. Also, the grammar is required to be context-free, which imposes restrictions on the files that can be translated.

The Ontos OODBMS has mechanisms to provide a uni-

fied view of data residing in different databases. In particular, Ontos allows product descriptions to be transferred between manufacturing systems and Ontos databases. Following the STEP standard, the structure (schema) of product data is specified using the EXPRESS language [4, 10], which is translated into an Ontos schema. Any product data file is an instance of a given EXPRESS schema. Users annotate such a file with path expressions on the corresponding schema, which can then be translated into an Ontos object. Although adequate and/or desirable in manufacturing applications, having the users annotate every single data file with the appropriate schema information would have been a serious problem in the ZOO environment. That's why we followed the Frog/Turtle approach.

Finally, SAS [6] and SPSS [7] are established statistical tools that allow file structures to be described by FORTRAN-like commands for statistical analyses, report writing, and data-file building. They are more powerful than Frog/Turtle because they can handle arbitrary processing of data as it is translated. On the other hand, as with all the tools mentioned in this section, they are not as easy to use as Frog/Turtle, since they do not offer a visual style of interaction.

10 Conclusions

We have presented Frog and Turtle, two tools that can be used to facilitate the translation of objects in an OODBMS into file format, so that files can be generated and sent as input to external programs. Frog is used off-line to map classes in OO schemas to areas of sample input files and specify how database values are to be laid out and printed in a file. The result is a map-file with all the appropriate information. Turtle is used at run-time to translate given database objects based on the map-file generated by Frog. Although we did not describe this at all in this paper, the same mechanisms are used for mapping (sub)-schemas to files that are to be read in by the OODBMS and translated into database objects. The only difference is that, in this case, we deal with reading specifications (not printing), which occasionally contain more entries, e.g., for set objects.

There are several issues that are part of our current and future work. First, the implementation of the second version of the system continues. Second, as described in this paper, all blocks in the map file contain path expressions rooted at the source class being mapped. This is quite inefficient, since these path expressions share long prefixes, which are processed again and again. We are currently investigating various ways to speed up the entire process. Third, we plan to investigate if printing specifications could potentially contain inconsistencies, in which case, algorithms should be developed for checking map-files for such inconsistencies.

References

[1] M. Chipperfield et al. Growth of data in the genome data base since ccm92 and methods for access. In *Proc. Human Genome Mapping*, pages 3–5, 1993.

[2] J. Cushing et al. Object-oriented database support for computational chemistry. In H. Hinterberger and J.C. French, editors, *Proc. 6th International Working Conference on Statistical and Scientific Database Management*, Zurich, Switzerland, September 1992.

[3] J. Cushing, D. Maier, M. Rao, D. Abel, D. Feller, and D. DeVaney. Computational proxies: Modeling scientific applications in object databases. In *Proc. 7th International Conference on Statistical and Scientific Database Management*, Charlottesville, VA, September 1994.

[4] ISO Group. ISO 10303: Guidelines for the development and approval of STEP application protocols, version 1.0. Technical Report ISO TC184/SC4/WG4NS4(P5), ISO, February 1992.

[5] E. Haber, Y. Ioannidis, and M. Livny. Opossum: Desktop schema management through customizable visualization. In *Proc. 21st International VLDB Conference*, pages 527–538, Zurich, Switzerland, September 1995.

[6] SAS Institute Inc. *SAS Language Reference - Version 6*. 1990.

[7] SPSS Inc. *SPSS User's guide, 3rd edition*. 1988.

[8] Y. Ioannidis, M. Livny, and S. Gupta. The ZOO desktop experiment management environment, February 1996. Submitted for publication.

[9] C. Lewerentz and E. Casais. STONE: A short overview. Technical Report FZI.040.1, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, May 1992.

[10] D. Schenck. Exchange of product model data - part 11: The Express language. Technical Report TC184/SC4 Document NC64, ISO, July 1990.

[11] J. Uhl et al. The object management system of STONE - OBST release 3.2. Technical Report FZI.027.1, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, October 1991.

[12] J. Wiener and Y. Ioannidis. A Moose and a Fox can aid scientists with data management problems. In *Proc. 4th International Workshop on Database Programming Languages*, pages 376–398, New York, NY, August 1993.