The *-Interface

Len Wanger

April 6, 1993

Disclaimer

This is still in the early draft stage. In particular, message and script formats and parameters are not very well fleshed out.

1 The Tioga front-end/back-end interface

Figure 1 shows a conceptual diagram of the communications between the front and back-ends in the Tioga system. All of the communications between the two ends of the Tioga system (i.e. the five arrows) are provided in a protocol called the **-protocol*.

The main process of the front-end is called the *visualization executive* (VE). The VE's main functions are to start up all of the front-end sub-processes, set up the communications between the front and back-ends, and to send/receive messages for the front-end.

To understand why the designated areas of *-protocol functionality are necessary it is useful to have a high-level understanding of what happens within the front-end. Within the VE there are three front-end sub-systems that need to communicate with the back-end (Figure 2).

The first sub-system is the *visualization planner* (VP). The VP automatically generates recipes which convert data into *renderable forms* (a.k.a. *renderables.*) The VP receives *descriptions* of incomplete recipes from the RM and returns plans for completed recipes in the form of *recipe plans*. *Recipe descriptions* are recipe plans that include a description of the recipe's ingredients as well as a characterization of the data flowing into and out of the instance of the recipe. This characterization of the data flowing between ingredients is necessary to allow *polymorphic ingredients*. Polymorphic

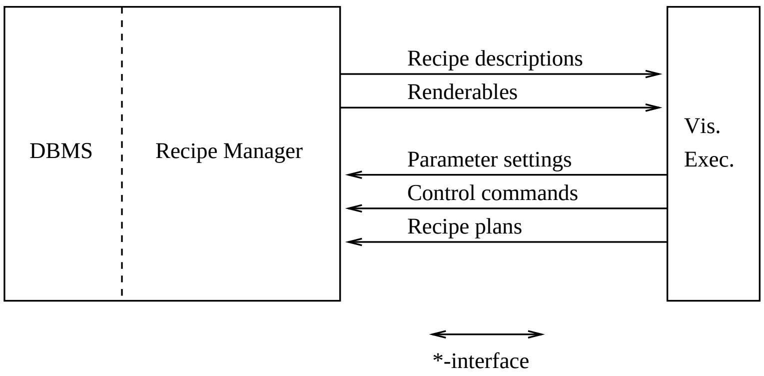

Figure 1: The conceptual organization of the *-interface.

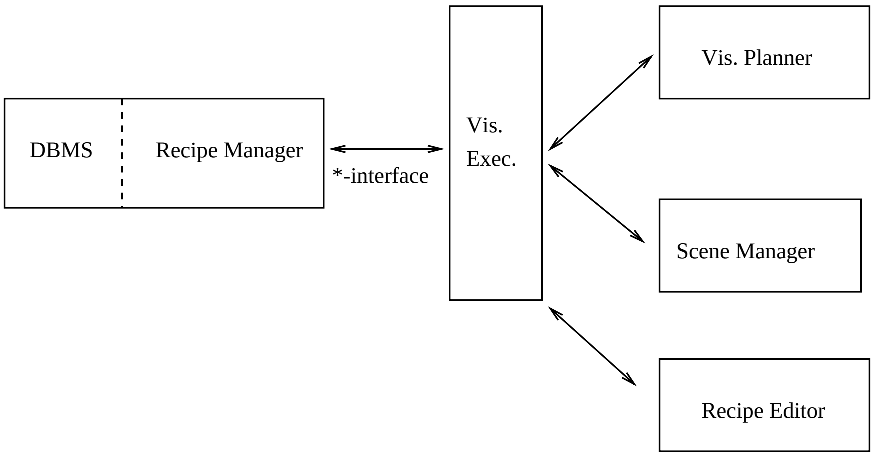

Figure 2: The conceptual organization of the front-end.

ingredients allow one ingredient to be used for several different data-types. The extract scalar ingredient is a good example of a polymorphic ingredient. Instead of writing separate pressure, temperature, and foo scalar extractor ingredients, a single scalar extractor ingredient can be written. The contextual information needed by the VP about the ingredients usage is provided by the data characterization.

Recipe plans are high-level descriptions of complete recipes that can be interpreted to construct actual recipes for execution by the back-end. Although recipe plans can specify modifications to existing recipes, the VP always returns plans for a complete recipe.

The second sub-system is the *scene manager* (SM). The scene manager maps renderable forms into screen visuals and maps user inputs on these visuals into actions. Renderable forms are device independent objects suitable for direct conversion into visuals. In addition to visual aspects (geometry and appearance attributes), renderable forms include behavioral actions for responding to user input. Renderable objects should be thought of as *widgets* (geometry + visual attributes + behavior). Renderable forms are created by the last stage of a complete recipe and are passed from the RM to the VE. Returned to the RM are the results of the application of behaviors in the form of recipe scripts.[1] Although most of these behaviors will result in small modifications to existing recipes, such as ingredient parameter settings, some behaviors may result in the specification of new recipes. For instance, double-clicking on a glyph representing a USGS weather station may open a window containing a visualization of the data for that station. This new window would depict the output from the execution of a new recipe spawned as a result of the user action, which may, or may not, require the assistance of the VP for completion.

The third sub-system is the *recipe editor*. The recipe editor is used to interactively create and modify recipes. The recipe editor can either create new recipes or receive existing recipes from the back-end in the form of recipe descriptions. The results of editing functions are sent to the back-end as recipe plans. These recipe plans can also include control commands for the back-end.[2]

[1] The mechanism for behaviors to be acted out is not yet determined. Potentially actions consist of parameter changes, new recipe generation, database queries, and storage/data manipulation commands.

[2] The recipe editor is being developed by Koko Nathan at Berkeley.

2 Issues

- **Object level interface**

  The *-protocol, which is meant to replace libpq and portal buffers, should work at the object level as opposed to the Postgres portal buffer level. This requires a *generic data model (GDM)* which should be used throughout the recipe manager. Having all recipe inputs and outputs in this GDM allows the visualization planner to interpret data at any point in the dataflow network. This is an important function of the visualization planner as described above. Since the implementation of the data model encapsulates the structure of the data in the model, it can be used in a strongly typed recipe system. The GDM is described in more detail below. Providing a powerful library of GDM access functions will also simplify the programming of ingredients. The data model to be used, *Fiber bundles*, is described below.

- **Scripting languages**

  Recipe plans and recipe descriptions should be specified in the scripting language. The scripting language should be usable for recipe descriptions, recipe plans, and parameter settings. The syntax of the language has not been determined, but will probably be similar to the scripting languages found in other visualization packages such as AVS and Data Explorer.

- **Separation from the database**

  The *-interface should be high-level and independent of any particular DBMS. Independence allows the front-end to interface to any DBMS, and the back-end to interface to any rendering system. DBMS independence also provides flexibility for future changes to be made transparently to the DBMS and visualization systems.

3 The *-interface

This section describes the function calls supported by the **-interface library.* The *-interface library consists of a set of functions called by the front-end to drive the back-end. Although it is assumed that the *-interface library uses fiber bundles as its data model, the *-interface does not include any commands to directly operate on fiber bundles.[3]

Note: The status returned by some of the function is a value indicating whether the call was successfully completed or not.

3.1 Function calls

- **General Messages**

  - **Start session**

    Status InitBE(<knowledge>, . . .);

    This function causes the back-end to initialize a new session. This initialization will include any initialization of the back-end needed as well as sending any necessary data to the front-end. An examples of this data would be for the RM to send the data for the knowledge base to the VP.

    What parameters are needed? There should be parameters sent and/or data returned for all of the initialization information.

  - **End session**

    Status exitBE();

    This function causes the back-end to terminate a session. Clean up may involve the storage of data into the DBMS. For example the VP may wish to store knowledge gained within the session to the knowledge base.

    What parameters are needed? There should be parameters sent for all information to be stored into the database.

- **Execute a recipe script**

  Status execScript(<recipe script>);

  This function executes a recipe script. Scripts can be used to modify recipes and execute control commands in the back-end. Recipe

[3] The commands proposed in the original *-interface to operate on data are replaced by data model methods associated with the fiber bundles. The old *-interface move, mark, and fetch commands can be viewed as iterator methods on the data model.

modifications can range from a simple parameter change to the complete specification of a new recipe. All of the instructions within a single recipe script should be performed as an atomic operation (i.e. no recipe calculations should be performed until the script execution is complete.) Details of the recipe scripting language appear below.

- **Describe a recipe**

  RecipeDesc descRecipe(<recipe ID>);

  This function returns the description of a recipe from the back-end. The description is in the form of a recipe script. Details of the recipe scripting language appear below.

  A recipe ID is some way of identifying from which recipe to get a description. What should this be?

- **Get data**

  FB getData(<recipe ID>);

  This function is used by the front-end to poll the back-end for output data from a recipe. The back-end returns either an output data set or an indication that no data is yet available. An output data set is returned as a collection of renderables contained within a fiber bundle (see the generic data model below).

  What parameters are needed here? How is no data indicated?

4 Recipe Scripting Language

This section needs to be completed.

- **Add module** Specify a module in the recipe.

  module <name>;

- **Set parameter** Set the value of a parameter in a module.

  parmSet [-range <min> <max>] <module>:<parm> <value>;

- **Connect port** Connects an output port to input port. Ports can be NULL to disconnect one module from another.

  portConnect <module from>:<port> <module to>:<port>;

Note: The recipe scripting language does not directly support data model access methods. Instead, data manipulations are performed by specifying recipes that contain data access methods within their ingredients.

How to associate data with the recipe?

5 Generic Data Model

This section needs to be completed.

An object-oriented data model has three functions:

1. to hold the data
2. to describes the structure of the data
3. to provide methods to operate on the data

The Tioga data model is an object-oriented data model based on the concept of fiber bundles. The Tioga implementation of fiber bundles contains the following parts:

- a base space - The space that the fibers lie in. This is the range of independent variables.[4]
- fiber bundle sections - The dependent data. Each point in the base space has a fiber associated with it. Fiber sections are collections of fibers. In the Tioga implementation each fiber is a "record". This implementation of fiber bundles is inherently hierarchical as fields of these records can themselves be fiber bundles.
- a structure description - A description of the structure of the fiber bundle written in a *data description language*. This is self-descriptive information enabling the interpretation of the structure and content of the data.
- methods - Access methods (functions) to examine and manipulate the bundles and their components.

[4]The mathematical concept of fiber bundles includes a *base space transfer function* that maps each point in the base space to a point in the fiber space. The base space transfer function is implicit in the instance of the base space in this implementation.

5.1 Base Spaces

- **list** (ordinal) - this is a list of ordered points. Each member of this ordered set is an association with a key (label) and data (the fibers). Lists are ordered (sorted) by providing a comparison function. Single points are treated as lists with one element. The only time single data points should be seen is when the data with the point is very large like a satellite image in which case the overhead is negligible. Pure nominals (unordered point sets) are treated as lists whose order is arbitrary (although probably the order the members were added.) Specialized ordinals could be added for alphabetic and numeric (ascending and descending) orders.

- **mesh** - this is a set of points within a topology. Each point has a key (a label or spatial position), a list of neighbors (topological information), and data (the fibers). Point sets can be represented as spatial points with no neighbor information (disconnected points in a space). Unstructured grids can be represented as either vertices with neighboring vertices (edges) or as faces with neighboring faces.

  Mesh is not a good term since it implies edges! Other names that don't work here are tessellation and sampling.

- **lattice** - a regular mesh. The regularity of the mesh means that topological information is implicit and does not have to be explicitly specified. Examples include rectilinear or hexagonal grids. The nodes of a lattice can be defined in cartesian, polar, spherical, cylindrical, etc. coordinates.

There will also have to be pre-defined base spaces for common data-types. Among these data-types are renderables (polygons, vectors, volumes, etc) and domain specific data (GCM , AVHRR, EOS , etc.)

5.2 Fibers

Fibers can be simple data-types such as strings, scalars, vectors, or they can be records. These records contain multiple fields, where each field can be of any data-type including a fiber bundle.

5.3 Structure Description

To be Completed.

5.4 Access Methods

To be Completed. See Butler and Pendley?

Access methods include:

- Constructors
- destructors
- reader
- writer
- dumper (debug)
- render
- iterators,

5.5 Implementation

To be completed (the 10,000 dollar question)!

Each fiber is a record. These fields of these records are fields of any data-type including nested fiber bundles. This way a fiber bundle is like an ADT.

The "of" relationship is a natural way of framing fiber bundles. A bundle would be defined as a <base space type> of <fiber type>. For instance a *list of string*, a *lattice of vector*, or a *mesh of foo*. In this last case foo could be a user defined data-type (a record or FB).

To show how fiber bundles are used we show several ways to encode bitmaps using Tioga fiber bundles. Bitmap data might look like:

define bitmap { lattice(xsize,ysize) of record { float r,g,b } };

Another organization is as three separate arrays each containing a color band:[5]

define bitmap { lattice(3) of (lattice(xsize,ysize) of float) };

RLE[6] encoding might look like:

define bitmap { lattice(1) of (list of record { int count; float r,g,b;}) };

Many logical organizations (and many more illogical organizations) of bitmaps using Tioga fiber bindles exist. These examples are only meant to illustrate three ways that it could be done.

5.6 Benchmark

The following is a partial list of data-types that have to be represented in Tioga. It is not meant to be complete, only to form a benchmark to show that the fiber bundle model is general enough to handle all of the data needs of the Tioga system.

- relational data
- bitmaps
- polygonal meshes
- time series data
- volumetric data (multi-dimensional arrays)
- directed and undirected graphs

All data passed will be in the form of the fiber bundle data model. Implementation details have not been determined.

[5]These examples are not a complete specification. For instance, there needs to be a place to hang the contextual information such as some way to name or label each band in the bitmap image (in this case red, green, and blue)

[6]Run length encoding (RLE) is a method of image compression that takes advantage of coherence. Each row in an RLE encoded images is represented as list homogeneously colored runs of pixels. Each run is a tuple of (c,n), where c is the color of the pixels in the run, and n is the length of the run in pixels.

6 Two simple scenarios to test the protocol

In order to demonstrate how the *-protocol works two simple scenarios are presented. These scenarios are designed to be very pragmatic - they are; quite simple, exercise most of Tioga's functionality, and represent mandatory functionality. Both scenarios are presented in the context of the scenario data presented by the Scripps researchers in the Sequoia 200 Scenario by Riddle et al. Of particular interest in these examples is the flow of control commands and data using the *-interface.

What is the control and data flow for these scenarios. Who calls who? What changes what?

All sessions start out as:

1. user starts VE
2. VE starts up VP, SM, and RE. User initializes RM session and starts the *visualization benchtop.* The visualization benchtop is the initial browser used when visualizing data.
3. user uses the browser to specify data-sets, recipes, and/or stored visualizations.
4. at some point the user indicates to start the visualization. This may be implied through the browser or by explicitly activating a submit button in the recipe editor.
5. the recipe editor determines whether the current recipe is complete (does it end in a display box?). If the recipe is not complete, the recipe description is sent to the VP. The returned recipe plan is executed resulting in either a complete recipe or a VP failure.
6. If the recipe is valid then the recipe is executed. If not, a visual indication is given to the user (the exact indicator is still unknown.)
7. The front-end starts grabbing fiber bundles containing renderables from the back-end. These renderables in these bundles are interpreted by the SM and displayed to the user as visuals on the display.

The scenarios take off from the point of a visuals appearing on the screen. The scenarios show what happens as a result of user interactions.

These scenarios are not complete. The scenario is defined but the actions taken to implement the scenario are not yet specified.

- **Scroll bar**

  The visual is a window containing a 2D line graph of maximum daily temperature over an interval of time. At the bottom of the window is a horizontal scroll bar. When the left scroll bar is picked, the time range for the graph changes (i.e. the starting and ending dates are decremented). Consider this scenario for cases where the data is cached locally (i.e. only a front-end change) or for when more data must be retrieved from the DB. Also, how are the data parameters within the recipe modified?

- **Pop up chart**

  The visual is a set of markers representing USGS weather stations overlaid on a map of the Sierra Nevadas. When one these station glyphs is picked a graph (similar to the graph in scenario I) appears. How is this graph created? How is this graph placed? Was this graph an invisible part of the recipe all along? Does this new graph have to generate a request to the visualization planner for a new recipe branch? How is it determined what information associated with the particular station is shown?

A Glossary

Display ID

Recipe see *dataflow network* or *network*

Network description descriptions of the instantiation of a network sufficient to create a network plan within the visualization planner. The description must include the network structure as well as contextual information about the individual network modules (from the database schema) and a description of the data being used in the network. Network descriptions are written in the network description language.

Network manager

Network plan network scripts created by the visualization planner to generate renderables from database objects. **Network script** a description of a network sufficient for the construction of the network in the network manager. network scripts are written in the network scripting language.

front-end

back-end

Module

Ingredient see *module*

visualization planner

widget

B Tioga - as specified in the Tioga document

The *-interface is:

- Mark NDP with identifier
- ERASE identifier
- MOVE to identifier
- MOVE to NDP
- MOVE to F(value) <operator> <constant> along (delta1,....,deltan)
- FETCH(number)

- FETCH (delta1,....,deltan)
- FETCH (delta1,....,deltan,angle)

plus two (the only ones specified) special DBMS registered functions:

- display (object, NDP, screen requirements)
- update()

Note: NDP stands for N dimensional polygon.

In the Tioga paper the only supported data type in the star interface is the NDP. However, it is clear from the examples given that minimally bitmaps have to be directly supported also.

The model of the front-end visualization system, is that of eye processes (visualization renderers) that request renderable output from the recipe manager by using * commands. All direct DBMS commands would use the standard libpq/PostQUEL interface through a separate connection than the recipe manager. At the last PictureTel meeting it was also agreed upon that some commands to allow input (interaction) will be needed. This was referred to as adding hands to the eyes model.

It was also agreed upon that the *-interface as specified in the Tioga document is not complete and that SDSC should propose extensions to this interface to provide us with the necessary functionality.

C Unanswered Questions

- What control commands are needed for the RM? Since the Recipe Editor is part of the front-end and will be going over the *-interface to the back-end, any functionality needed by the Recipe Editor must be available through the *-interface. Since all user-interface component are in the front-end the *-protocol does not provide any interface support. SDSC will provide Koko with a way to get this information that is local to the front-end.
- Are time contract negotiations wanted/needed?
- It is unclear whether database queries will always be cast into recipe ingredients, or whether they will be directly applied to the database.