A Visual Query Language for Software Visualization

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI 02912 spr@cs.brown.edu
401-863-7641, FAX: 401-863-7657

Abstract

Visualization for software understanding requires that the user be able to define specific visualizations that are specialized to the particular understanding task at hand. In this paper we describe a system that lets the user define such visualizations quickly and effectively. The system uses a visual query language over a variety of data sources to let the user specify what information is relevant to the understanding task and to correlate that information. Then it provides a mechanism for letting the user select and customize an appropriate visualization for this data.

1. Problem Definition

Software understanding is the task of answering questions and gaining insights about software systems. In some cases it involves gathering broad-stroke information about what a system is doing; in other cases it involves understanding a particular aspect of the system; in still other cases it involves addressing very specific issues such as why was this routine called or what will happen if I change this input.

Software visualization for understanding must address the basic issues that understanding implies. It must provide access to multiple data sources in order to address the wide range of applications implied by software understanding. It must provide the means for integrating and relating these sources so that the complex questions that arise in software understanding can be readily addressed. It must permit a wide range of questions, from the very narrow to the very broad. Moreover, it must let the user get a visual answer to an understanding problem quickly and accurately, typically providing a detailed solution in about five minutes.

To facilitate software understanding, a front end of software visualization must provide a variety of facilities and meet a number of requirements. The most important of these are:

- It must provide a query language over multiple data sources. If the available data is viewed as a (possibly virtual) database, then asking questions over this data is essentially querying that database.
- It must provide independence from the underlying data formats and structures. Programmers are already burdened with the complexities of the language, environment, and the particular system. It is not reasonable to make them learn the wide variety of formats that will be used in collecting and analyzing the data that is needed for understanding.
- It must allow the easy addition of new data sources. As systems get more complex, new data sources become relevant. As software understanding problems become more complex, more data sources and data analyses are needed to address them. Any system should be able to easily accommodate such changes.
- It must support a variety of different formats of data. Some of the data will be available in relational databases. Other data might be available as linked objects, for example symbol tables or abstract syntax trees. Other data might be available in XML format, for example program analysis data through GXL [5] or the results of various analysis tools. Other data will be available dynamically through requests of existing systems, for example configuration data can be obtained dynamically using appropriate commands to CVS or RCS.
- It must provide full query capabilities. The questions that arise in software understanding can be quite involved and the data sources can be quite complex. In order to get the proper answers from these sources, the front end must provide a powerful and flexible query language.
- It must make common queries simple. While many understanding problems are complex and unique, a significant number of the queries that need to be addressed occur repeatedly. Such repeated queries, no matter how complex they are, should be easy to ask.
- It must be intuitive and easy to use. Programmers are harried enough and are loathe to learn or use a new tool. A visualization tool that is not easy to use will simply not be used. Programmer's don't yet understand the potential value of visualization and will not take the

time to learn a difficult or obscure tool. A consequence of this is that ideally the front end should be integrated into an existing programming environment.

- It must provide easy integration with a visualization system. The front end should let the user define the question that needs to be addressed and, at the same time, should let the user define what the visualization should look like.

We are currently developing a comprehensive system for using visualization for software understanding [8]. This system, BLOOM, includes facilities for collecting both static and dynamic data about systems [7], facilities for analyzing that data in various ways [9], a wide variety of visualization styles [10], facilities for browsing over, interacting with, and correlating the visualizations, and a general facility for defining what should be visualized and how it should be displayed. It is this last facility, MURAL, provides a powerful visual query environment that is the topic of this paper. This package attempts to offer a powerful front end for the user that meets the above criteria.

The next section describes the visual query language that MURAL provides for specifying what data is relevant to a particular understanding problem. The following section describes how MURAL supports a wide variety of different data sources, mapping them appropriately into entities and relationships. The next section describes the approach that MURAL takes to finding a visualization once the user has defined the relevant data. The final sections describe related work, experiences to date, and future plans for the system.

2. The MURAL Query Language

MURAL provides a visual query language based loosely on entities and relationships. Entities here represent the various data sources. Relationships represent ways of correlating these data sources, either though pointers, indices, or arbitrary associations. The visual front end shows the entities as boxes and the relationships as arrows connecting the boxes as can be seen in Figure 1.

2.1 Entities and Relationships

Entities are used to represent objects. They can be used to represent a tuple from a relational database, an object from an object database, an element from an XML file, or a simple C++ or Java object. Each entity consists of a set of fields that are defined over a basic set of domains. The domains include primitive domains such as strings and integers, references to other entities, and named instances of these domains. Entities in the visual editor actually represent sets of such objects.

Relationships represent ways of relating the entities in one set with those in another. Relationships can be simple, i.e. based on fields in one entity that contain pointers (either direct or indirect) to an instance of another entity. For example, a static definition entity contains a field labeled source that points to the file entity in which that definition occurs. Relationships can also be more complex. The set of entities representing classes can be indexed by the full class name. Thus, any entity that contains a field with a class name can be related to this set of classes through this index. Relationships that are even more complex are possible. The user can relate to entities by indicating that a certain set of fields should have common or correlated values.

The user can create multiple relationships between two entities. When this is done, the user is prompted as to whether these should be merged into a single relationship, hence representing an AND of the conditions of the original relationships, or if they should be left alone, representing an OR of the relationship conditions.

Figure 1 shows an example query using MURAL. This particular query combines profiling statistics accumulated for each pair of calling-caller routines along with static information about those routines and their classes. Entities in the query are represented as boxes containing a set of fields. Each field is labeled with the field name and data type. Fields that contain internal data such as pointers are not displayed. Relationships are labeled and are indicated by arcs between the entities. Either the labels are the name of the field that is used for simple relationships based on pointers or indices, or a user defined name for more complex relationships. Triangles at the end of the arcs are used to indicate the direction and arity of the relationships.

2.2 Combined Entities

To facilitate complex queries and the definition of visualizations, MURAL introduces several concepts. The first is the notion of *combination*. Entities and relationships can be combined into new entities or relationships. Two or more entities along with the relationships that connect them can be combined into a single entity. The result is similar to taking a join of the underlying sets using the relationship as the join expression. For example, Figure 2 shows the result of selecting the entities *TfileMethod* and *TfileClass* entities for the *from* relationship along with their accompanying *class_id* relationship and combining them into a single entity labeled *MethodClass*.

Combined entities are useful in a variety of ways. Their primary use is to let the user define new data objects as logical combinations of existing objects based on the relationships among the existing objects. This is necessary for defining what should be visualized in our overall frame-

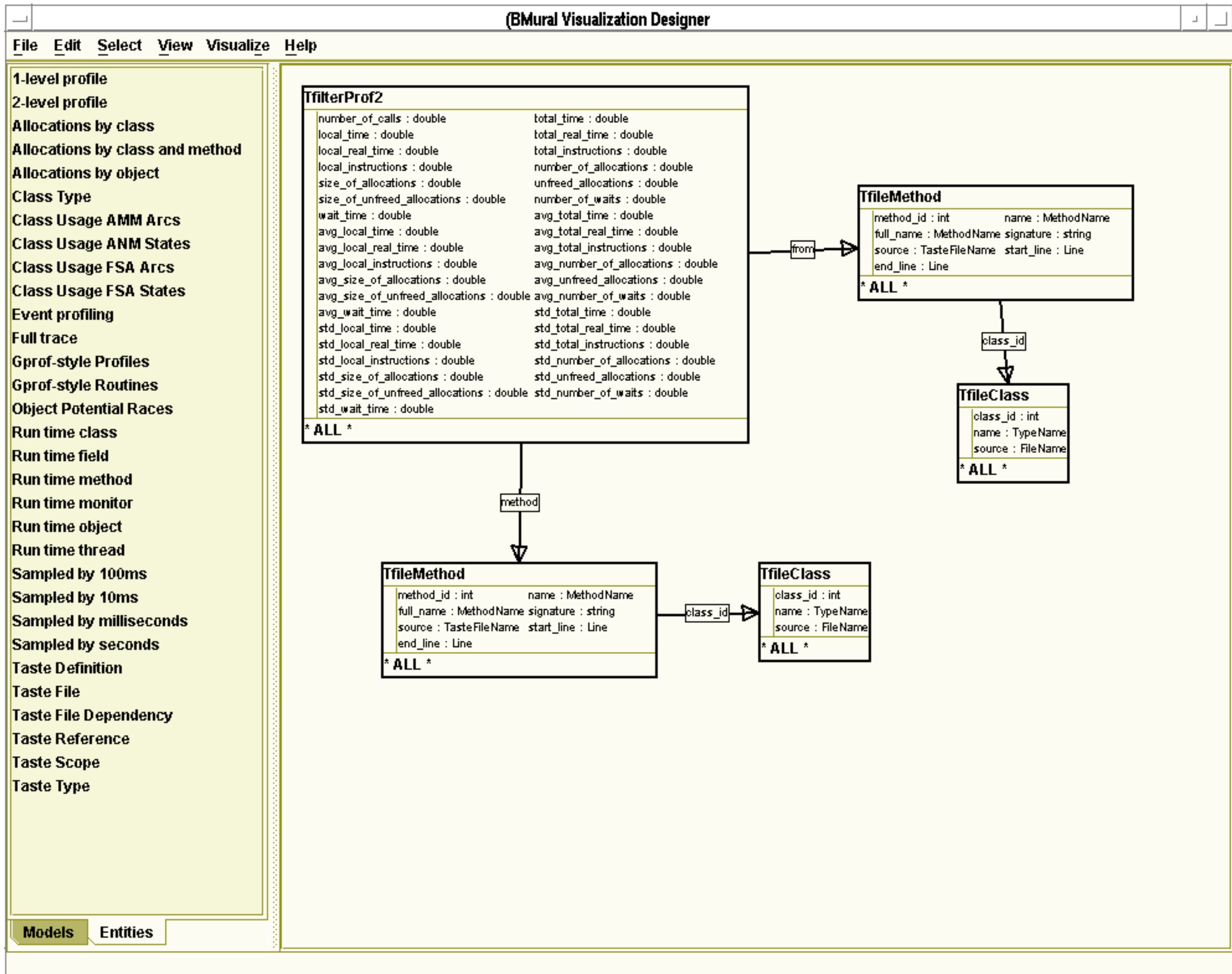

FIGURE 1. Snapshot of MURAL showing the setup of a query that correlates 2-level call analysis (statistics about routine A calling routine B) with static information about the two routines and their classes.

work. Second, they make the specification of complex queries simpler by providing levels of abstraction. A combined entity essentially is an abstraction of the various entities and relationships that went into it.

Combining entities also provides the means to define queries involving transitive closure. MURAL treats the case where two identical entities are combined using a single relationship as special. It asks the user to specify the range of times, from zero to infinite, that the relationship can be followed. This provides a natural and intuitive way of both expressing and denoting transitive closure operations in the visual query. Note that transitive closure is essential to queries in the software domain since it is needed for asking about class hierarchies, scope containment, and call containment (A calls routines that eventually call B).

2.3 Combined Relationships

Just as multiple entities can be combined using the relationships between them to form new entities, a series of relationships and their intervening entities can be merge to form a combined relationship. The implication here is that the data in the selected entities is not needed, just the relationship. For example, Figure 2 also shows the effect of combining the *method* and *class_id* relationships along with the intermediate *TfileMethod* class to create a *calling_class* relation.

Combined relationships provide many of the benefits of combined entities. They simplify both the visual image of a query and its definition by providing a level of abstraction. They let the user make explicit what relationships should be the focus of the visualization that is being defined. Also,

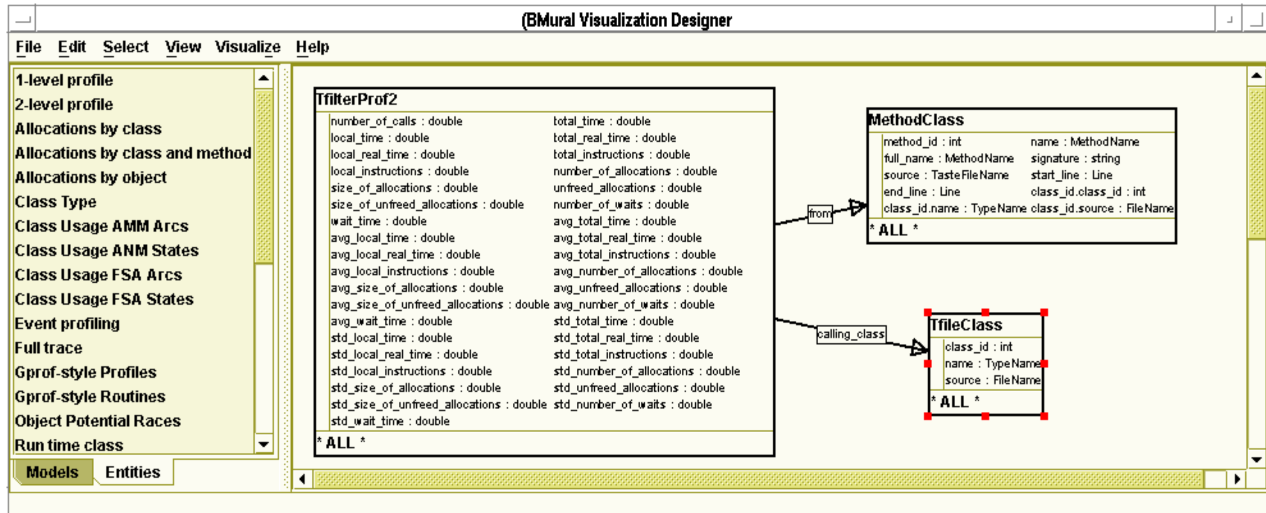

FIGURE 2. MURAL supports grouping operations. Here the *Method* and *Class* entities that are separate in the original query are combined into a single entity for the *from* relationship while the *method* and *class_id* relationship along with the *TfileMethod* entity are combined to form the *calling_class* relationship.

they make it easier to use logically AND together multiple relationships.

2.4 Restrictions and Fields

The second concept that MURAL uses to facilitate complex queries is the notion of restrictions. The user can define arbitrary restrictions on each entity in the form of Boolean expressions. These restrict which objects in the set implied by the entity actually are represented by the entity on the display. Restrictions can be defined either on base or combined relationships. Note that by defining restrictions on combined relationships, the user is actually defining more complex, cross-entity conditions. This is another way that combined entities allow easier and more intuitive definition of sophisticated queries.

The third concept that MURAL uses is to let the user restrict and modify the set of fields that an entity contains. Individual fields can be removed either from the visual display of the query or from consideration in the construction of the visualization or both. This can be used to save screen space and to make large and complex entities more understandable while building queries. It is also useful in defining a restricted and possibly more relevant set of fields for a particular visualization. This is needed to let the user specify what is important in the visualization that is ultimately being defined.

In addition to limiting the set of fields, MURAL lets new fields be defined for an entity. This is done by providing a new field name and an expression over the other entity fields that provides the new field's value. Computed fields let the user define new data elements that should be used in the visualization. They also allow the stepwise definition of restrictions and relationships since the new fields can be used to define new restrictions and relationships.

2.5 Submodels

In addition to making it possible to construct complex queries, MURAL attempts to make it easy to ask common queries. Once a query is constructed, it can be saved as a submodel. Then the user can instantiate such submodels as needed when building the query. When the most common queries are stored as submodels, such queries can be invoked with a simple series of clicks. For example, Figure 3 shows the query of Figure 1 defined as a submodel. The initial display provided by MURAL shows the user the set of available submodels to encourage the user to select an existing model if one is relevant.

2.6 Expressive Power

The query language that results is at least as powerful as the relational algebra or calculus. The basic operators that need to be provided to accomplish this are product, select, and project. The query language handles products through relationships in general. While most relationships express restricted products or joins, the use of arbitrary, field-based relationships allows the definition of arbitrary products if these are needed. Selects can be done either through relationships which imply a selection of the corresponding product, or through restrictions on the entities. Projects are done by limiting the set of fields in the entity or by eliding entities or relationships when doing a combination opera-

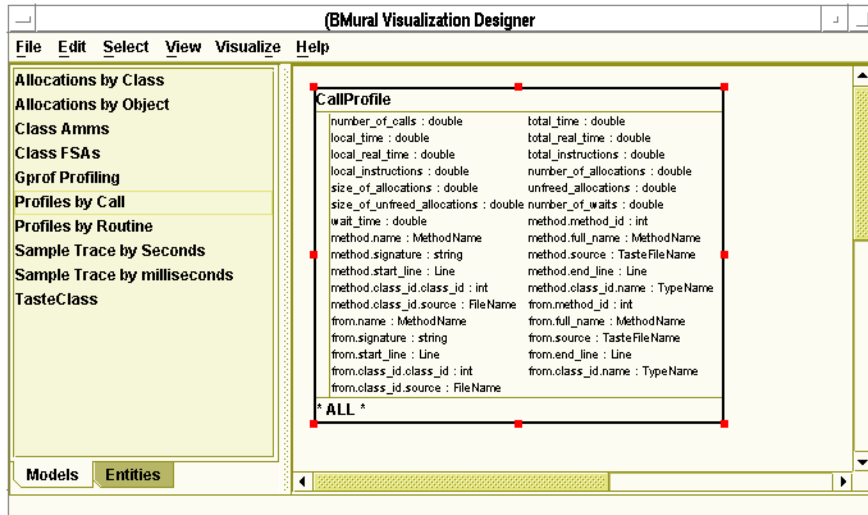

FIGURE 3. The call-profile query of Figure 1 shown as a submodel. This view also shows the current set of available submodels which cover the most common visualizations that a user might want to see.

tion. Note that this limitation does not affect the fields that are available for restriction or for relationship definition.

The query language offers additional capabilities that are only found in extended relational databases, logic database, or object databases. The ability to treat the combination of a class with another instance of the same class using a relationship as a transitive closure provides the ability to define transitive closure-based queries that cannot be defined in a relational framework.

3. Handling Multiple Data Sources

One of the more powerful aspects of MURAL is that it can handle a wide and extensible variety of data sources. It does this through the use of a common data model based on entities and fields and the ability to define arbitrary data sources in terms of that model. The query language reflects this data model rather than the characteristics and restrictions of the original data sources.

The model starts with a notion of domains of data to provide a common basis for multiple data sources. Each domain represents a basic type of data such as integer or string. Domains can be arranged hierarchically with lower levels of the hierarchy acquiring additional semantics. Thus a subdomain of string is *filename* which represents a valid filename string and a subdomain of integer is *line* which represents a line in a file. Domains are also used to represent references, either direct pointers or indirect unique identifiers, to a particular entity.

XML-based definitions are then used to define each data source in terms of entities and fields. A sample from such a file is shown in Figure 4. These let the user define new domains as references to existing domains, as subdomains of existing domains, or as arrays of existing domains as shown by the top definitions in the figure. The user can also define new entities representing information from a data source. Each data source can have multiple such entities associated with it. Each entity specifies both how to access the data source and the fields associated with the entity. The example shown represents the result of trace analysis providing statistics for all allocations grouped by class. The analysis is obtained from an XML file which is produced by running the TFILTER program, and each entity corresponds to a CLASS element in the XML file. This information is described in the ACCESS element of the domain definition. The various fields of the entity are each given a name and a domain and rules for obtaining that field from the source XML file either from an attribute or from a related element.

This scheme has been used for accessing information from XML-based analysis as above as well as from a relational database, from a specialized repository of attributes maintained by the trace collection package, and from an in-memory Java object database of static information gathered by a programming environment. The back end of the system takes care of creating efficient access methods to the different data sources based on the actual resultant query [10].

4. Specifying Visualizations

Once users have selected the data of interest by choosing an appropriate set of entities constrained by a combination of relationships and restrictions, they will want to get a visualization of the result.

```
<DOMAINS>

<DOMAIN NAME="TfileClassRef" TYPE="REF" REFER="TfileClass" />
<DOMAIN NAME="TfilterThread" TYPE="NAMED" DOMAIN="ThreadName" />
<DOMAIN NAME="TasteTypeRef_Array" TYPE="ARRAY" BASE="TasteTypeRef" />

<DOMAIN NAME="AllocClass" DESCRIPTION="Allocations by class" TYPE="ENTITY">TTIP="Allocation statistics organized by class">
    <ACCESS TYPE='XML'
        RUN="$(TFILTER) -d $(TRACEDIR) -AC"
        FILE="$(TRACEDIR)/AllocClass.tf"
        ELEMENT="CLASS" />
    <FIELD NAME="class" DOMAIN="TfileClassRef" XMLATTR="ID" />
    <FIELD NAME="number_of_objects" DOMAIN="int" XMLATTR="OBJECTS" GROUPING="Sum" />
    <FIELD NAME="number_freed" DOMAIN="int" XMLATTR="FREED" GROUPING="Sum" />
    <FIELD NAME="average_span" DOMAIN="double" XMLATTR="TIME" GROUPING="Sum" />
    <FIELD NAME="average_moves" DOMAIN="double" XMLATTR="MOVE" GROUPING="Sum" />
    <FIELD NAME="average_gcs" DOMAIN="double" XMLATTR="GC" GROUPING="Sum" />
    <FIELD NAME="stddev_span" DOMAIN="double" XMLATTR="TIMESD" GROUPING="Sum" />
    <FIELD NAME="stddev_moves" DOMAIN="double" XMLATTR="MOVESD" GROUPING="Sum" />
    <FIELD NAME="stddev_gcs" DOMAIN="double" XMLATTR="GCSD" GROUPING="Sum" />
  </DOMAIN>

<DOMAINS>
```

FIGURE 4. A sample entity definition. This definition represents statistical information about allocations by class. It is obtained from an XML file using the specified access rules and XML information specified for each field. The *class* field contains a reference to a *TfileClass* object.

MURAL provides an internal mechanism that finds appropriate visualizations and offers the user an ordered choice of what it deems appropriate for viewing the specified data. This mechanism starts with definitions of each of the available visualization strategies. These definitions provide MURAL with a model of what type of data are used in the visualization, a brief description of the visualization, the set of parameters that control the visualization, and the set of visualization fields to which the data described in the user's query can be mapped.

An example of such a definition is shown in Figure 5. The definitions are in two parts. First the definitions specify the set of data domains that are used in the various visualizations. Each of these is defined as a set of possible source domains.

The second part of the figure shows the definition of our spiral visualization. The first part of this definition describes the object model. This particular visualization is driven by a set of objects each of which must have a from and a to time fields. These fields must have a data type that corresponds to the visualization domain *Index*. In addition, there are optional visualization fields for the height, width, color and texture associated with the visualization. More complex models allow the specification of multiple object types and relationships between these types.

When the user finishes building the data model, he clicks the visualization button in MURAL. At this point the system attempts to match the user's model with each of the visualization data models. The matching attempts to associated each entity with a visualization object type in such a way that all required fields are present and as many of the optional fields as possible are available. Multiple entities can be mapped to a single visualization object type.

The matching process is actually more complex than this. In doing the matching, not only does MURAL consider the explicit user entities, but it also considers possible operations on those entities including:

- Combining multiple entities using intervening relationships as is supported by the query language.
- Combining multiple relationships into a single relationship as is supported by the query language.
- Merging two identical entities into a single entity using a union operation. This is needed to handle self-loops in the visualization model since such loops are not directly representable in the query model.
- Omitting entities and relationships by not associating them with any visualization entity or relationship.

The system uses heuristics to assign a cost to each of these operations and a value to each of the matching fields. The result of these values is used to sort the list of potential visualizations for the given user data model. This list is pruned by eliminating strategies that are not within 50% of the best selection or within 90% of the best selection for a given visualization.

5. Related Work

The visual query language itself is built on top of numerous other efforts aimed at providing visual database interfaces. These start with Query by Example [15] which defined a table-based interface. Other examples include PSQL [11], various Entity-Relationship query languages [4,12,14], SeeQL [13], G+ [2], DOODLE [3], and

```
<VIZDATA>

<DOMAINS>
<DOMAIN NAME="Color" COLOR="TRUE">
      <USE DOMAIN="int" />
      <USE DOMAIN="long" />
      <USE DOMAIN="float" />
      <USE DOMAIN="double" />
      <USE DOMAIN="string" />
      <USE DOMAIN="enum" />
      <USE DOMAIN="flag" />
    </DOMAIN>
</DOMAINS>

<VISUALIZATIONS>
<VISUALIZATION NAME="SpiralFlavor" DESCRIPTION="Spiral" >
       <REQUIRES>
         <REQUIRE>
           <ENTITY NAME="Object">
             <FIELD NAME="FromTimeIndex" DESCRIPTION="From Time" DOMAIN="Index" MAP="Range
             <FIELD NAME="ToTimeIndex" DESCRIPTION="To Time" DOMAIN="Index" MAP="RangeMap"
             <FIELD NAME="ZFromIndex" MAP='LevelMap' DESCRIPTION="From Height" DOMAIN="Ind
             <FIELD NAME="ZToIndex" MAP='LevelMap' DESCRIPTION="To Height" DOMAIN="Index"
             <FIELD NAME="WidthIndex" MAP='WidthMap' DESCRIPTION= "Width" DOMAIN="Index" O
             <FIELD NAME="ColorSpec" DESCRIPTION="Color" DOMAIN="Color" OPTIONAL="1"/>
            <FIELD NAME="TextureSpec" DESCRIPTION="Texture" DOMAIN="Texture" OPTIONAL="1"
            </ENTITY>
          </REQUIRE>
         </REQUIRES>

</VISUALIZATION>
</VISUALIZATIONS>
</VIZDATA>
```

FIGURE 5. Sample visualization definition. The first part shows the definition of visualization domains as sets of possible data domains. The second part provides a description of our spiral visualizations first by specifying the data model to be visualized and second by defining the parameters of the visualization.

VISUAL [1]. Our query language differs from these both in its features and its focus. Most of these languages are aimed at specifying exact queries. Our language is oriented toward specifying a broad collection of data for visualization with the assumption that any specific data element will be isolated as part of the visualization process.

The systems that are closest to our work include Query by Diagram (QBD) [12]. QBD uses an entity-relationship base to construct relational queries, with both entities and relationships containing information. It starts with the overall ER database schema and lets the user select paths within that schema. It provides bridges to allow connections based on arbitrary conditions. It can translate loops into generalized transitive closure. Our system, on the other hand, does not assume an initial entity-relationship schema or even use a true entity-relationship database model. Instead, it lets the user connect entities that represent data sets dynamically. Our relationships are essentially links rather than the traditional ER-type relationships. Moreover, QBD has no equivalent to our use of combinations of both entities and relationships.

We also used experiences from our previous work where we used a much simpler query visual language based on a universal relation assumption that became too bulky and complex for specifying relevant queries [6]. Rather than pursuing this previous model, we used the lessons we learned in making the MURAL query language both more powerful and simpler.

The handling of multiple data sources builds on the broad body of work in federated databases. Our work is different in that it is designed to handle a wide range of data sources beyond databases as well as database-related sources. At the same time, we have not done a lot of the optimization work that previous systems do to make federated queries run fast. The front end described here assumes that there is an efficient back end to obtain and visualize the data. We have a start at that back end, but not one that handles arbitrary queries very efficiently.

Finally the work on finding appropriate visualizations is based loosely on the various efforts at finding appropriate data display formats for different types and combinations of data.

6. Experience and Future Work

The MURAL system goes a long way toward meeting the requirements for software visualization we previously defined. It provides a common model of multiple data sources that hides the nature and structure of the source and its data from the user and makes it easy to accommodate new or additional sources. It provides a full query lan-

guage oriented toward specifying the data and relationships that should be visualized. This language makes use of the notion of combination to let the user quickly build new data objects and relationships that should be the basis for the desired visualization. The system uses submodels to make specifying common visualizations simple. Moreover, the language is designed to be intuitive and easy to use. Finally, the whole framework is designed so that the system can select the appropriate visualization and thus integrate easily into the overall visualization framework.

We have been using a fully-functional MURAL as a front end for visualization only for a limited time and for a limited set of users (the back end is not totally complete) and thus our experience to data is limited. We have demonstrated, however, that it is possible to define a wide range of different visualizations quite easily using the front end. Moreover, these visualization typically interrelate data from several different sources, demonstrating the utility of a general purpose query language. The visualization range from views of the stack during execution to views of allocations by class and method to views of the statics structure of the file and class dependencies of a system.

The problems we have had with the query language are two fold. First, restrictions and user-defined operations are still a little difficult to use and are not adequately reflected in the visual representation. Second, we have found a need to be able to specify aggregate queries. Right now aggregation is available in the back end as a feature of the visualization engine. It seems logical and some users have asked that it also be available as part of the query language.

We have not had any problems with new data sources and have been able to match all our available data to the entity-relationship form that MURAL provides. Adding new data sources to MURAL itself is relatively trivial. Adding new access methods to the back end to support these sources typically takes one to two days of programming.

The major problem to date with MURAL itself has been the heuristics used in selecting the appropriate visualization. Our experience here is that the best visualization is always one of the ones chosen as appropriate, but is generally not the one that is chosen as most appropriate. We are currently looking into determining a better set of heuristics and a better set of values for selecting the best visualization.

Overall, our experiences with MURAL have been quite positive. As we complete the rest of the visualization system and make it easy to obtain and then view data about program behavior and structure, we hope to make the whole system available to a larger and broader range of users. This will provide the real test of whether the query language we have developed and described here is the appropriate one.

7. Acknowledgements

This work was done with support from the National Science Foundation through grants ACI1025046, CCR1039110, and CCR9702188 and with the generous support of Sun Microsystems. Parts of the code were written or modified by Joshua Levin; significant advice and feedback was provided by Manos Renieris.

8. References

1. N. Balkir, G. Ozsoyoglu, and Z. Ozsoyoglu, “A graphical query language: VISUAL,” Case Western Reserve University (1997).

2. I. F. Cruz, A. O. Mendelzon, and P. T. Wood, *Proc. 2nd Intl. Conf. on Expert Database Systems*. 1989.

3. I. F. Cruz, “DOODLE: a visual language for object-oriented databases,” *ACM SIGMOD Intl. Conf. on Management of Data*, pp. 71-80 (1992).

4. R. Elmasri and J. Larson, “A graphical query faciliy for ER databases,” *Proc. 4th Intl. Conf. on Entity-Relationship Approach*, pp. 236-245 (October, 1985).

5. Richard C. Holt and Andy Schurr, “GXL: toward a standard exchange format,” *Workshop Conference on Reverse Engineering 2000*, (November, 2000).

6. Steven P. Reiss, “Software visualization in the Desert environment,” *Proc. PASTE ’98*, pp. 59-66 (June 1998).

7. Steven P. Reiss and Manos Renieris, “Generating Java trace data,” *Proc Java Grande*, (June 2000).

8. Steven P. Reiss, “An overview of BLOOM,” *PASTE ’01*, (June 2001).

9. Steven P. Reiss and Manos Renieris, “Encoding program executions,” *Proc ICSE 2001*, (May 2001).

10. Steven P. Reiss, “Bee/Hive: a software visualization backend,” *IEEE Workshop on Software Visualization*, (May 2001).

11. Nicholas Roussopoulos and Daniel Leifker, “An introduction to PSQL: a pictorial structured query language,” *Proc. IEEE Workshop on Visual Languages*, pp. 77-87 (1984).

12. Guiseppe Santucci and Pier Angel Sottile, “Query by Diagram: a visual environment for querying databases,” *Software Practice and Experience* Vol. **23**(3) pp. 317-340 (1993).

13. Bosco S. Tjan, Leonard Breslow, Sait Dogru, Vijay Rajan, Keith Reick, James R. Slagle, and Marius O. Poliac, “A data-flow graphical user interface for querying a scientific database,” *IEEE Symp. on Visual Languages*, pp. 49-54 (August, 1993).

14. Z. Q. Zhang and A. O. Mendelzon, P. Ng, and R. Yeh, “A graphical query langauge for entity-relationship databases,” in *Entity-Relationship Approach to Software Engineering*, ed. S. Jajodia,North-Holland (1983).

15. M. M. Zloof, “Query by Example: a data base language,” *IBM Systems J.* Vol. **16**(4) pp. 324-343 (1977).