AAAI Technical Report FS-12-03
Discovery Informatics: The Role of AI Research in Innovating Scientific Processes

Capturing and Using Knowledge about the Use of Visualization Toolkits

Nicholas Del Rio
University of Texas at El Paso

Paulo Pinheiro da Silva
Pacific Northwest National Laboratory

Abstract

When constructing visualization pipelines using toolkits, developers must understand what sequencing of operators will transform their data from its raw state to some requested visual representation. In some cases, the requested visual representation must be generated from hybrid pipelines, composed of both toolkit-based and custom operators. Traditionally, developers learn about how to construct these visualization pipelines by word of mouth, by reading documentation and by inspecting code examples, all of which can be costly in terms of time and effort expended.

The Visualization Knowledge Project (VisKo) is built on a knowledge base of visualization toolkit operators including rules for how operators are chained together to form pipelines. VisKo helps scientists by automatically generating and suggesting fully functional visualization pipelines, alleviating scientists from having to write any pipeline code. This paper reports on the kinds of knowledge required to support automatic pipeline generation as well our successes when applying VisKo to a number of visualizations scenarios spanning geophysics, environmental and materials science.

Introduction

Visualization toolkits such as Generic Mapping Tools (GMT) (Wessel and Smith 1998), Visualization Toolkit (VTK) (Schroeder, Martin, and Lorensen 1998), and NCAR Command Language (NCL) (NCAR 2012) provide sets of visualization specific functions (e.g., format and data type conversion), known as *operators*, from which scientists chain together to form visualization pipelines. Visualization pipelines are operator sequences that transform data from its raw form into graphical or textual representations that are more easily digested by scientists. In an ideal setting, employed visualization toolkits would support documentation that address concerns of scientists, including understanding whether toolkits: (1) are capable of generating requested visualizations, (2) are capable of ingesting input datasets in formats they naturally reside, (3) are equipped with sufficient documentation describing the behavior of each supported operator and how they can be combined.

Each unmitigated concern has consequences pertaining to what toolkit scientists may choose to employ and how much effort they may have to expend to construct required pipelines. If (1) is not addressed, then scientists must look to another toolkit to generate the required visualization. If (2) is not addressed, then scientists are left with the task of corralling their data into formats that that toolkit can ingest, which may involve writing custom code or reusing operators from other packages that perform the required transformations. If concern (3) is not addressed, the pipeline development process may fall back to trial-and-error, which can be very time consuming considering the number of different operator sequence combinations.

In this paper we present the Visualization Knowledge Project (VisKo http://trust.utep.edu/visko) that provides an approach for automatically constructing visualization pipelines provided a knowledge base of visualization toolkit operator capabilities. The VisKo knowledge base is a formal, reusable, and machine readable alternative to typical toolkit manuals written in natural language, e.g. English. By capturing and using knowledge about the use of visualization toolkits, VisKo can overcome the limitations of constructing visualization pipelines when concerns (1)-(3) are difficult to mitigate.

The paper begins with a description of how to manually configure pipelines that generate volume renderings and highlights the knowledge needed by scientists to construct these kinds of visualization pipelines. The next section describes the VisKo ontology set and how it captures and structures the knowledge necessary to construct a pipeline with the use of VTK. The paper then describes the VisKo system, which leverages a knowledge base of visualization toolkit operators in order to automatically construct pipelines. The paper then follows with a report on our success with applying VisKo to a number of different fields ranging from environmental to material sciences. We then follow with a discussion of the scientific impact as well as future challenges faced with maturing VisKo. Finally, the paper closes with related work and conclusion.

Generating Volume Visualizations using VTK

VTK is a toolkit suited for generating 3D visualizations. For the pipeline example in this paper, we leverage VTK to generate volume renderings of a 3D seismic model of a subterranean region of the Earth. Our seismic models are simple 3D grids of velocity values with the following properties: the 3D grid is encoded as a flat binary array stored in lit-

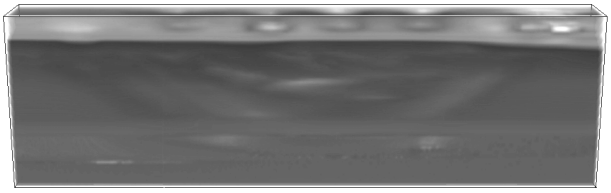

Figure 1: Volume Rendering of Velocity Model

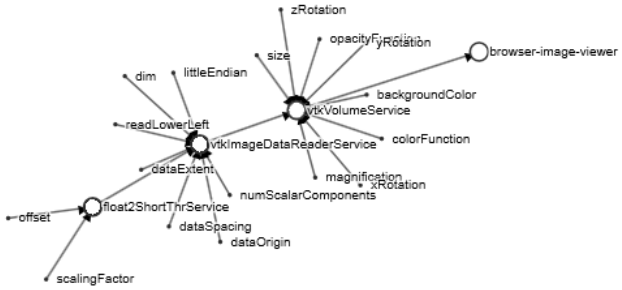

Figure 2: Volume Rendering Pipeline

tle endian, each grid cell stores a single scalar, each scalar represents a floating point value. From velocity models, scientists can learn about the density and geologic composition of a region. Figure 1 presents a volume rendering of a seismic velocity model. The *holes* or *gaps* in the image are a result of missing seismic information.

The VTK-based pipeline that generates the volume is presented in Figure 2. In this graphical pipeline notation, all the larger circles, except for the final circle without any outgoing edges, represent visualization toolkit operators while the arrows represent the flow of data from one operator to the next. For example, the output of `float2ShortThrService` flows as input to the next operator in the sequence, in this case `vtkImageDataReaderService`, and so on and so forth until the data has been transformed into a JPEG image that can be presented by the `browser-image-viewer` (i.e., a Web browser represented by the circle without any outgoing arcs). Additionally, the pipeline reveals the parameter sets of each operator denoted as dots feeding into the operators, such as `offset` and `scalingFactor` parameters of `float2ShortThrService`.

In order to construct the pipeline in Figure 2, scientists must know that `vtkVolumeService` is an operator that generates the required volume geometry and associated retinal properties (e.g., color, opacity, orientation), thus addressing concern (1) in the introduction. In terms of information visualization, `vtkVolumeService` represents a `Mapper`, or an operator that is able to extract presentational information, such as geometry, color, and spatial orientation, from data that is otherwise devoid of this information. In addition, scientists must know that `vtkImageReaderService` is a suitable operator for converting the raw velocity model (i.e., the 3D grid encoded as a binary array) into `vtkImageData` format, an encoding of the 3D grid that can be consumed by `vtkVolumeService` and thus addressing concern (3).

However, `vtkVolumeService` is highly configurable and can be set to use different kinds of ray casting algorithms to generate volumes. If `vtkVolumeService` is employing a `vtkVolumeRayCastMapper`, as is the case in our example, then the `vtkImageData` that is ingested must consist only of short integers. Our input velocity model is composed of floating point values and so our pipeline must begin execution with a custom `float2ShortThrService` operator, which transforms the raw velocity model into an array of short integers thus addressing concern (2). It is knowledge about these kinds of restrictions that we want to capture in our VisKo knowledge base that we believe can save scientists considerable time and effort when developing visualization applications. Additionally, having access to a framework that allows for the reuse of custom non-toolkit specific operators, such as `float2ShortThrService`, would also reduce costs associated with development.

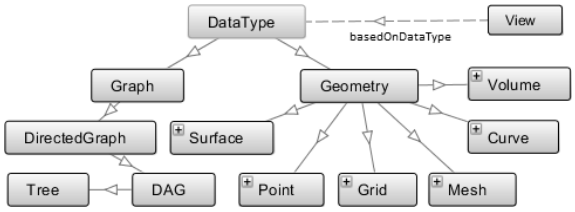

Figure 3: View Ontology

VisKo Visualization Ontologies

The goal of our VisKo knowledge base is to enable machines to automatically compose visualization pipelines, which traditionally are constructed manually. We therefore need to encode the *visualization knowledge* that can be extracted from toolkit manuals into formal statements that can be processed by reasoning agents configured to construct pipelines. We have proposed to encode the visualization knowledge using the Web Ontology Language (OWL) (Hitzler et al. 27 October 2009), which can be processed by existing open source inference engines such as Pellet (Sirin et al. 2007). Our resultant ontology is decomposed into three sub-ontologies *VisKo-View*, *VisKo-Operator*, and *VisKo-Service*, which specialize in the kinds of views and data types that can be generated, the operators that generate those views and data types, and the services that implement the operators respectively. The set of ontologies can be downloaded from http://trust.utep.edu/visko/ontology.

VisKo-View Ontology

One of the first inquiries a scientists will have when considering a toolkit is what views can the toolkit generate. A *view* in this context refers to a data type along with presentational information such as color, size and any textual adornments. The view ontology, presented in Figure 3, defines a class `View` and captures relationships between views and data types through the property `basedOnDataType`.

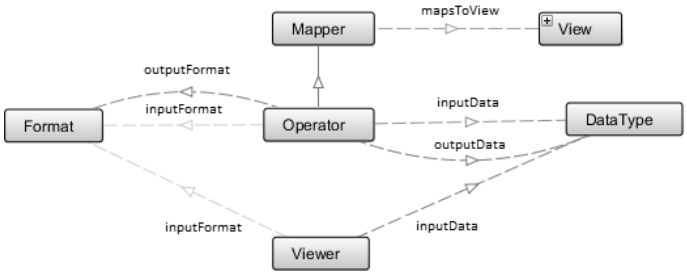

Figure 4: Operator Ontology

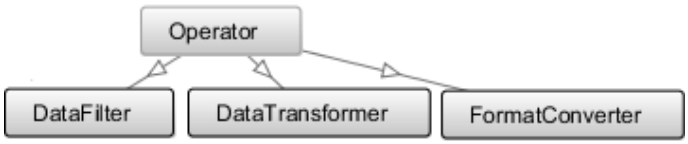

Figure 5: Inferred Operator Concepts

Additionally, the ontology classifies data types based on whether they are graph or geometric based, where `Graphs` are typically reserved for information visualization scenarios and `Geometries` are more commonly used in scientific visualization. The solid directed arcs refer to the `hasSubClass` property, indicating that the target class is a specialization of the source class. To further refine the class `Geometry`, we imported geometric data types from an ESIP datatype ontology (ESIP 2007), which includes but is not limited to: `Points`, `Surfaces`, `Volumes`, `Grids`, `Meshes` and `Curves`. We will see in the next section how we can describe the behavior of mappers, such as `vtkVolumeService`, by referencing particular views.

VisKo-Operator Ontology

We have already encountered two kinds of operators in the volume visualization pipeline presented in Figure 2: the `float2ShortThrService` format converter and the `vtkVolumeService` view mapper that generated the volume view. In addition to the operators `FormatConverter` and `Mapper`, the VisKo-Operator ontology captures two other types of operators: `DataFilter` and `DataConverter`.

Although there exist four distinct operator subclasses in VisKo-Operator, scientists only assert instances of a few select classes. Figure 4 presents the VisKo-Operator ontology from the perspective of scientists, which presents the classes `Viewer`, `Operator` and its subclass `Mapper`. Additionally, this subset of the ontology also defines the properties used to the describe operators' and viewers' interfaces, which are defined in terms of the input and output `DataType` and `Format`. Some data formats are capable of encoding a multitude of different data types, such as netCDF (UCAR 2012) that encode arrays of arbitrary dimensions. This flexibility is the reason our operator interfaces pair format descriptions with the data types they encode. `Mappers`, in addition to supporting input/output type and format properties, are also associated with the property `mapsToView` that specifies the particular view the mapper is responsible for generating. The `DataType` class is defined in VisKo-View, previously presented, while the `Format` class is borrowed from the Proof Markup Language (PML-P) provenance ontology (McGuinness et al. 2007).

Figure 2 also presented the `browser-image-viewer`, which introduced the notion of a viewer. In Visko-Operator, the `Viewer` class is responsible for presenting a particular view to the screen and in our ontology, presented in Figure 4, the interaction between scientists and viewers is not modeled. The most we can assert about a viewer is its input interface, which is also defined in terms of input type and input format. This is evident with the `browser-image-viewer`, which has an ingoing arc (i.e., data flowing in) but no outgoing arcs.

Although the few constructs presented in 4 allow us to describe the interfaces of operators in enough detail to support automatic chaining, scientists might want further information about operators' roles. We can automatically elaborate on the classification of operators by inspecting their input/output data type and format interfaces. Figure 5 presents the three additional subclasses that can be used to further classify instances of the generic class Operator. `DataFilters` can be identified because they are operators that have the same input and output interface (i.e., the same input/output data type and format). This suggests that no structural changes to the data are made but that only some form of sub-setting or filtering is applied. `DataTranformers` are operators that have differing input and output data types. For example, most gridding algorithms such as minimum curvature or near-neighbor are considered transformers because they convert their irregularly distributed input 2D point data into 2D grids. Finally, `FormatConverters` can be identified by checking if the input and output formats differ while the input and output data types remain identical. `float2ShortThrService` is an example of a converter because only the encoding (i.e., the format) of the 3D grid being processed is altered.

VisKo-Service Ontology

Only interface requirements of operators can be described using the VisKo operator ontology. However, the Visko-Service ontology represented in Figure 6 augments these operator descriptions with information about *where* and *how* to invoke services that implement these interfaces. The execution details are described in terms of the Web Ontology Language Service (OWL-S) ontology (David Martin 2005), which defines a comprehensive set of classes and properties needed for describing executable Web services. The `implementsOperator` is our property that associates OWL-S service descriptions with operators.

There are benefits to keeping invocation details about operators separate from the more abstract operator descriptions. Using Visko-Operator, we can describe an operator `gridding`, which ingests 2D point data encoded as an XYZ ASCII table and generates a 2D grid encoded in netCDF (UCAR 2012); using our classification rules, we would infer that this `gridding` operator is of type

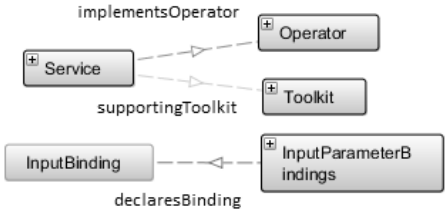

Figure 6: Service Ontology

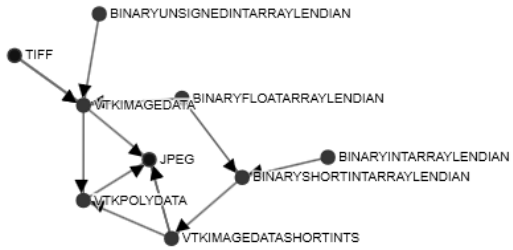

Figure 7: Format Transformation Paths

`DataTransformer`. In GMT (Wessel and Smith 1998) there are two implementations of this gridding data transformer, `grdContour.exe` and `surface.exe`, which each support a different gridding algorithm. Using VisKoService, we can state that both these gridding services implement the single gridding transformer, thus informing our system that both `grdContour.exe` and `surface.exe` can fulfil the abstract operator requirements.

The `InputParameterBindings` class models a list of parameter bindings (i.e., a parameter and associated input value). Typically, operators composing visualization pipelines are heavily parameterized and the `InputParameterBindings` allows us to specify a set of hard-coded values that can be fed into the operators in the case when scientists are unsure what values to use. The challenge of working with parameters is elaborated on in the discussion section.

Other concepts defined in Figure 6 include `Toolkit`, which represents the different visualization toolkits such as GMT, NCL, and VTK. `Services` can be associated with a toolkit and this information is useful if scientists want to filter pipelines by what toolkit supports them.

Composition Rules

The portions of the VisKo ontologies presented thus far are only used to describe the operators and services of a toolkit but do not contain knowledge about how to compose these operators into pipelines. For this task, VisKo relies on OWL2 (Hitzler et al. 27 October 2009) property chains to identify possible pipelines given our knowledge base. Consider our search space as a graph where nodes represent `Format` instances and edges indicate that formats `canBeTranformed` to another format. In order for a `canBeTransformed` edge to exist between format nodes *A* and *B*, there must exist an operator that has `inputFormat` *A* and `outputFormat` *B*. Thus our search graph contains all possible combinations of format transformations where only a subset of the paths may yield meaningful visualizations. This graph is just one way in which we can view our VisKo knowledge base that is focused only on the perspective of format transformations, as shown in Figure 7. Although we must also consider data type transformation paths, the approach is identical to analyzing format paths and so its description is excluded from this document.

Given our format transformation graph, one natural task is to check whether some format can be transformed into another. In the case of our velocity model volume visualizations, we need to check if BINARYFLOATARRAYLENDIAN (i.e., the format velocity models are encoded in) can be transformed into a JPEG image, the format our `Browser-Image-Viewer` can consume. We employ transitive OWL2 property chains to make all transformations explicit so that we can pose simple queries to answer these kinds of questions. The following horn rules represent these transitive OWL2 property chains in a more human friendly format than the RDF/XML format they reside in our ontology.

isFormatInputTo(?Fmt1, ?A) :-	inputFormat(A, Fmt1)
canBeTransformedTo(?Fmt1, ?Fmt2) :-	isFormatInputTo(Fmt1, A), outputFormat(A, Fmt2).
canBeTransformedTo(?Fmt1, ?Fmt2)	is a transitive rule

The first rule, `isFormatInputTo` is a simple inverse property of the asserted `inputFormat`. `isInputFormatTo` serves as an antecedent to the `canBeTransformedTo` rule, which is transitive and states *Fmt1* can be transformed to *Fmt2* if there is a service that has an input format of *Fmt1* and an output format of *Fmt2*. Since this rule is transitive, we can infer if *Fmt1* can be transformed to another *Fmt3* though a sequence of one or more transformer operators. OWL2 inference engines such as Pellet (Sirin et al. 2007) can be triggered to apply the composition rules to the asserted knowledge base and infer the `canBeTransformedTo` relationship between all applicable formats. The derivation trace (i.e., proof) that Pellet uses to infer *canBeTransformedTo* relationships contains the order of operators needed to transform our input format into a target format.

The VisKo Knowledge Base In Use

Before our VisKo system can generate visualization pipelines, a knowledge base of toolkit operators must be constructed. This is a challenging step because it requires that some human or automated scraper extract information from toolkit manuals and encode this information as triples conforming to our VisKo ontology set. Once we have our asserted model of toolkit operators, we can apply the composition rules and generate an inferred model from which we can use to automatically generate visualization pipelines.

Subject	*Predicate*	*Object*
float2ShortThrService	is a	Operator
float2ShortThrService	inputFormat	BINARYFLOATARRAYLENDIAN
float2ShortThrService	outputFormat	BINARYSHORTINTARRAYLENDIAN
float2ShortThrService	inputDataType	3DGrid
float2ShortThrService	outputDataType	3DGrid
vtkImageDataService	is a	Operator
vtkImageDataService	inputFormat	BINARYSHORTINTARRAYLENDIAN
vtkImageDataService	outputFormat	VTKIMAGEDATASHORTINTS
vtkImageDataService	inputDataType	3DGrid
vtkImageDataService	outputDataType	3DGrid
vtkVolumeService	is a	Mapper
vtkVolumeService	mapsTo	Volume
vtkVolumeService	inputFormat	VTKIMAGEDATASHORTINTS
vtkVolumeService	outputFormat	JPEG
vtkVolumeService	inputDataType	3DGrid
vtkVolumeService	outputDataType	3DGrid

Figure 8: Asserted Model

Figure 9: Inferred Model

Subject	*Predicate*	*Object*
BINFLOATARRAYLEND	canBeTransTo	BINSHORTARRAYLEND
BINFLOATARRAYLEND	canBeTransTo	VTKIMAGEDATASHORTINTS
BINFLOATARRAYLEND	canBeTransTo	JPEG
BINSHORTARRAYLEND	canBeTransTo	VTKIMAGEDATASHORTINTS
BINSHORTARRAYLEND	canBeTransTo	JPEG
VTKIMAGEDATASHORTINTS	canBeTransTo	JPEG

Constructing a Knowledge Base of Operators

Using the VisKo ontological concepts and properties, we can describe the VTK toolkit operators composing our volume generation use case. Figure presents our asserted toolkit model, describing the operators comprising the VTK pipeline in Figure 2. The statements below describe only a fraction of the VTK operator suite, which contains hundreds of operators. In practice, our current asserted knowledge base is hosted on GitHub (GitHub 2012) and contains 12 `DataTransformers`, 9 `Mappers`, 7 `Viewers`, 5 `Toolkits`, and 129 `Parameters`.

The statements in Figure are triples where, the operator in question (i.e., the subject) is described in terms of its relationship (i.e., predicates) to other objects. In our framework, these descriptions are actually encoded in RDF/XML and it is these descriptions that comprise our asserted knowledge base that the Pellet reasoner applies our composition rules to generate the inferred model presented in 9.

Visualization Queries

Provided an operator knowledge base of both asserted and inferred statements, we have developed a web application that generates pipelines which can generate visualizations requested by scientists. This system is known as VisKo (http://trust.utep.edu/visko) and accepts visualization requests specified in a query-like form we refer to as *Visualization Queries*. An example visualization query that requests for a volume visualization is shown in Figure .

The query requests that the dataset `vel.3d` should be visualized as a volume that can be viewed using mozilla-firefox, where the format of vel.3d is `BINARYFLOATARRAYLENDIAN`. Additionally, the query specifies that vel.3d contains a data type `Seismic-Velocity-Model`, a type that is a subclass of `3DGrid`. Our VisKo system will perform a series of steps to identify the pipeline required to generate the requested volume. Firstly, the system will issue a SPARQL query to determine whether the *source* format, in this case `BINARYFLOATARRAYLENDIAN`, can be transformed to some *target* format operated on by some viewer contained within `mozilla-firefox`, in this case a `JPEG`. If true, we can use the proof that Pellet used to infer the `canBeTransformed` to identify what sequences of operators are needed to support the transformation. VisKo will then remove all operator pipelines that fail the data type restriction, which ensures that the output data type of an operator must match the input data type of the next operator in a given sequence.

VISUALIZE	vel.3d
AS	volume
IN	mozilla-firefox
WHERE	FORMAT = BINARYFLOATARRAYLENDIAN
	AND TYPE = Seismic-Velocity-Model

Figure 10: Visualization Query for Volume Visualization

Run	Configure	Description
Run pipeline	Edit parameters	Text / Graph
Run pipeline	Edit parameters	Text / Graph

Figure 11: Format Transformation Paths

Once operator sequences that fail the data type restrictions have been removed, we can query for the service implementations of each operator and return the sequences of services back to the user as a pipeline result, shown in Figure 11. In this example, VisKo was able to generate two different pipelines that satisfy the visualization request, differing only in the output format of the resultant volume, PNG and JPEG. In some cases, VisKo is able to generate over 15 pipelines that satisfy a single query, by using different combinations of intermixed operators from different toolkits (e.g., toolkit hybrid piplines). From the pipeline result table, scientists have the options to (1) execute the pipeline, (2) set operator parameter bindings, and (3) browse a description of the pipeline in either a textual or graphical form such as the image presented in Figure 2.

Pipeline Execution

Upon selecting to run a given pipeline, VisKo will invoke each service in succession, similarly to how UNIX systems execute command-line pipes by forwarding the output of each service as an input to next service in the sequence. In our query example, the input data *vel.3d* is forwarded to the first service of the pipeline and is transformed until it is processed by the final operator, upon which a URL to the resultant visualization is presented to the user.

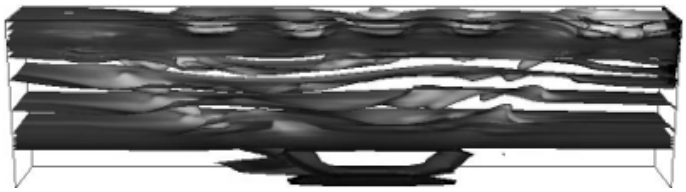

Figure 12: Isosurfaces Visualization of Velocity Model

Successful Applications of VisKo

The first application of VisKo was to generate GIS based visualizations of gravity data sourced from the PACES initiative (PACES 2002), where 2D plots, raster images, and contour maps were generated. More recently, VisKo has been employed to support a number of different scientific visualization scenarios, encompassing a number of different data formats, data types and views. We have used VisKo to visualize time series and scatter plots sourced from aerosol data gathered from NASA's Giovanni (http://disc.sci.gsfc.nasa.gov/giovanni/overview/index.html) as well as seismic velocity models generated from the CYBER-ShARE initiative at the University of Texas at El Paso (CYBER-ShARE 2012). In addition to the volume visualization presented in Figure 1, VisKo is also able to generate an isosurfaces rendering of the same velocity models presented in Figure 12, thus providing scientists with different views or perspectives of a single dataset. Finally, we have used VisKo to automatically coordinate the preprocessing stages of a materials science pipeline, employing tessellation generation services to construct a mesh that can be fed into a complex simulation.

Discussion

Scientific Discovery Enabled by VisKo

In VisKo, *scientific discovery* is enabled by the declarative nature of its visualization requests that do not require scientists to know all available data analysis capabilities to compose responses to these requests, i.e., build pipelines. This means that, for example, for a gravity contour map, measured data can be interpolated by either a near-neighbor or minimal curvature algorithm, even if the scientist was only aware of the availability of a minimal curvature interpolation service. In this case, it may be that the near-neighbor algorithm produces better results that may eventually lead to discoveries that were not enabled by the minimal curvature algorithm. Thus, VisKo may enable scientists to make new scientific discoveries and also learn about new data analysis capabilities.

Visualization Parameters

Properly setting operator parameters that yield useful visualizations is a very challenging task, and unfortunately VisKo prescribes no solution for this problem. Although VisKo can store preconfigured parameter values, these values are not guaranteed to be useful for every dataset. From our experience, we have observed that even with correct pipeline sequences, poor parameter values can yield empty visualizations, or even worse, misleading representations. For example, in our volume example, incorrectly setting the opacity function can hide many features that scientists may regard as interesting. Similarly, when generating isosurfaces, too many surfaces can create *noisy* images and setting too few can hide features in the data.

We have proposed to augment the VisKo pipelines with a prepossessing stage that may provide scientists with a good range of usable parameter values. Our first task was to break up the parameters into two classes, which we refer to as *data-driven* and *visualization-driven* parameters classes. Data-driven parameters are typically associated with the structure or *characterization* of data and include parameters such as dimensionality, scalar range, and byte-order. We believe that we can deploy services to inspect data and extract the values to these kinds of parameters. Values bound to visualization-parameters, which include opacity function, color functions, and isoline/surface intervals may be more challenging to infer due to the preferential nature of these values. Many of these parameters could be set if properties of the data, such as spatial or scalar value density distributions were known, but in general these values must be set by scientists themselves.

Related Work

Duke and Brodlie (Duke, Brodlie, and Duce 2004) proposed a visualization ontology that was initially sketched in a workshop report (Brodlie). In this work, they describe visualization in similar terms as VisKo: visual representations (i.e., views), techniques and renderings (i.e., operators), and services. Additionally, they describe how a visualization ontology might be segmented according to different concerns: World of Representation, World of Users, World of Data, and World of Techniques, which in our ontology correspond roughly to VisKo-view, visualization queries, types and formats, and VisKo-operator respectively. Duke and Brodlie (Duke, Brodlie, and Duce 2004) also speak of a separation of concerns between logical and physical layers, where logical layers may refer to our interface descriptions of pipeline operators in the Visko-Operator ontology and the physical layer may correspond to our OWL-S services referenced in Visko-Service.

However, to the best of our knowledge, Duke and Brodlie have not actually defined rules to aid in the automated composition of pipelines. Our approach relies on the progression of data through different formats and data types, which are not explicitly defined by Duke and Brodlie. Additionally, our model is founded on combining operators that can generate a visualization that some target viewer can display. This notion of a target viewer is absent in the Duke and Brodlie ontology.

Other authors have proposed models for the purpose of taxonomizing the set of available visualization techniques, such Chi's Data State Model (Chi and Riedl 1998; Chi 2002), which characterizes different visualization techniques according to how data is transformed from its raw *value* (i.e., initial state) to the *view* (i.e., final state). In the data state model, operators are classified according to what

state in the pipeline they operate in and thus require a white-box understanding of visualization applications. Of course, the goal of his work is to provide an understanding of the internals of visualization applications in order to be able to compare among techniques. In open world environments such as the Web, we need a more black-box approach to classification, such as the Visko-Operator ontology which classifies operators based on their input/output data type and format interfaces.

Conclusions

VisKo ontologies have been used to model visualization processes, providing a way for scientists to encode their knowledge about visualization toolkits and for machines to facilitate the scientists' task of building visualization pipelines. The paper described how visualization pipelines were automatically derived by OWL reasoners through the application of pipeline composition rules. These pipelines although conceptual, may have an executable binding, in which case VisKo provides a fully implemented infrastructure that automates the process of generating visualizations. We have shown that in the presence of these capabilities, scientists can declaratively request for visualizations using a query-like language without specifying any executable details such as what operator or services should participate in the generation of requested visualizations.

Acknowledgements

We would like to acknowledge NASA's Dr. Gregory Leptoukh who was a strong supporter and visionary our our work. We would also like to acknowledge the CyberShARE center (http://cybershare.utep.edu/) at the University of Texas at El Paso for funding support.

References

Brodlie, K. W. Visualization Ontologies. http://www.nesc.ac.uk/talks/393/vis_ontology_report.pdf.

Chi, E. H.-h., and Riedl, J. 1998. An operator interaction framework for visualization systems. In *INFOVIS '98: Proceedings of the 1998 IEEE Symposium on Information Visualization*, 63–70. Washington, DC, USA: IEEE Computer Society.

Chi, E. H. 2002. Expressiveness of the data flow and data state models in visualization systems. In *AVI '02: Proceedings of the Working Conference on Advanced Visual Interfaces*, 375–378. New York, NY, USA: ACM.

CYBER-ShARE. 2012. University of Texas at El Paso CYBER-ShARE Center. http://cybershare.utep.edu/.

David Martin, Mark Burstein, e. a. 2005. OWLS: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/.

Duke, D. J.; Brodlie, K. W.; and Duce, D. A. 2004. Building an ontology of visualization. In *Proceedings of the conference on Visualization '04*, VIS '04, 598.7–. Washington, DC, USA: IEEE Computer Society.

ESIP. 2007. Earth Science Information Partners ESIP Federation Datatype Ontology. http://wiki.esipfed.org/index.php/Data-Service-Ontologies.

GitHub. 2012. Github social coding. https://github.com/.

Hitzler, P.; Krötzsch, M.; Parsia, B.; Patel-Schneider, P. F.; and Rudolph, S., eds. 27 October 2009. *OWL 2 Web Ontology Language: Primer*. W3C Recommendation. Available at http://www.w3.org/TR/owl2-primer/.

McGuinness, D.; Ding, L.; Pinheiro da Silva, P.; and Chang, C. 2007. PML2: A Modular Explanation Interlingua. In *Proceedings of the AAAI 2007 Workshop on Explanation-aware Computing*.

NCAR. 2012. NCAR command language reference manual. http://www.ncl.ucar.edu/Document/Manuals/Ref_Manual/.

PACES. 2002. Pan American Center for Earth and Environmental Studies. research.utep.edu/Default.aspx?alias=research.utep.edu/paces.

Schroeder, W.; Martin, K. M.; and Lorensen, W. E. 1998. *The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Sirin, E.; Parsia, B.; Cuenca Grau, B.; Kalyanpur, A.; and Katz, Y. 2007. Pellet: A practical OWL-DL reasoner. *Web Semantics* 5:51–53.

UCAR. 2012. Network common data form netcdf. http://www.unidata.ucar.edu/software/netcdf/.

Wessel, P., and Smith, W. H. F. 1998. New, improved version of generic mapping tools released. *EOS Transactions* 79:579–579.