VisKo: Semantic Web Support for Information and Science Visualization

Nicholas Del Rio and Paulo Pinheiro da Silva

The University of Texas at El Paso, Computer Science,
500 W. University Ave. El Paso TX 79968 USA

Abstract. Specialized knowledge in visualization software packages such as Visualization Toolkit (VTK), Generic Mapping Tools (GMT), and NCAR Command Language (NCL) is almost always a requisite for writing applications that visualize datasets or information. Technical understanding of visualization packages including rendering capabilities and data format ingestion is needed before it can be determined whether a package can satisfy some set of visualization requirements. Even after identification of satisfactory visualization packages, an application must still be built on packages that generate the required visualizations.
Visualization Knowledge (VisKo) modularized ontology set encodes knowledge about visualization software packages using semantic Web technologies (e.g., RDF, OWL, and Pellet) in order to facilitate the scientist task of visualizing data and information. In the presence of VisKo, users may move away from writing visualization applications and move towards an environment where they can request for visualizations declaratively and without being fully aware of a wide range of visualization package implementation details.
The use of VisKo ontologies is supported by a fully implemented environment composed of a query language, a visualization knowledge base, and a set of visualization services. The VisKo ontology and supporting environment has been in use by many scientific initiatives including visualizing artifacts associated with a seismic tomography, gravity anomalies, and brightness temperature.

1 Introduction

Writing visualization applications requires a culmination of skills held by scientists, visualization experts, and engineers. Scientists may know what views (i.e., graphical analogues not relational database views) suit a particular data whereas visualization experts may know about third party visualization packages that generate and render required views. Once views are chosen and supporting visualization packages targeted, engineers then need to understand how to integrate visualization packages into custom applications that visualize datasets or information. This visualization knowledge associated with scientists, visualization experts, and engineers is what VisKo provides ontologies for encoding.

Applications built on existing visualization packages usually share some structure: data must be transformed into formats that can be ingested by the visualization packages, renderers must be configured by arguments sets, and finally,

rendered views may be further transformed into formats suitable for viewers (i.e., applications that present the visualizations to users). Once users have researched visualization software packages, experimented with prototype applications, and finally matured the prototypes into useful visualization applications, the knowledge gained from the experience of writing the application is either encoded into the application itself in the form of imperative code, or locked inside the minds of the users who developed the application. Although it can be useful to learn how to write visualization applications through inspection of example code, ideally the knowledge required to write visualization applications would be made available in some canonical form separate from executable code. With access to visualization knowledge encoded by VisKo ontologies, inferencing systems might be able to reuse the knowledge to automate the development of similar visualization applications or reconfigure existing applications for new purposes.

The paper highlights the VisKo ontology set in the context of a use case for visualizing gravity datasets described in Section 2. Section 3 introduces the concept of third party visualization packages known as a *toolkits* and describes how toolkit functions are assembled into applications known as *pipelines*. Section 4 presents the VisKo ontology and how it serves as a grammar of a canonical language used to describe different visualization toolkit pipelines. Section 5 discusses how we reason with toolkit descriptions encoded in VisKo language to answer visualization queries, which provide a declarative alternative for writing imperative code. Section 6 discusses some issues with our approach while Section 7 provides some related work. The paper concludes with Section 8.

2 Gravity Map Use Case

Gravitational anomalies or features are highlighted when data is visualized as a contour map or colored contour map because users can quickly identify where spikes exists by finding regions on the map containing a high diversity of colors or contour lines close in proximity. A contour map visualization of gravity data is presented in Figure 1 and a sequence of high level tasks in Figure 2 describes how these kinds of contour maps can be generated.

Gravity data is sourced from the Pan American Center for Earth and Environmental Science (PACES) data store [20], which provides access to datasets as a collection of XYZ records plus some metadata related to stations. XYZ data records can be broken down into two components: a 2D spatial coordinate (i.e., XY or longitude and latitude) and the associated scalar (i.e., Z) thus classifying gravity data as 2D. Section 3 describes the different kinds of third party visualization packages that render contour maps and describes how the high level tasks in Figure 2 are implemented as pipelines using the toolkits.

3 Visualization Toolkits, Operators, and Pipelines

Visualization packages or *toolkits* as they are commonly referred to, provide reusable sets of visualization functions referred to as *operators* [5]. Below is a list

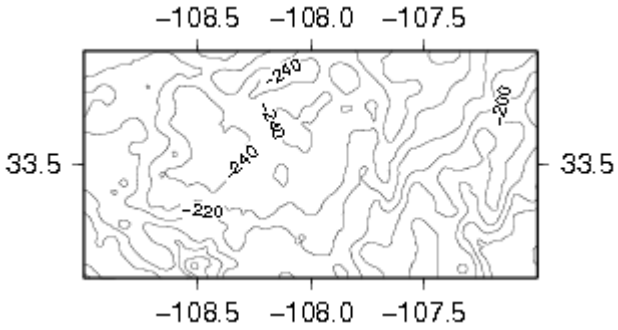

Fig. 1. Gravity Contour Map

Fig. 2. Task description on how to build a gravity contour map

Task	*Description*
1. Gather:	Gather the raw gravity dataset readings for the specified region of interest. The data provided by Pan American Center for Earth and Environmental Sciences (PACES) [20] gravity database is tabular and formatted in ASCII.
2. Filter:	Filter the raw gravity dataset readings by removing unlikely and duplicate point values. Unlikely values are those readings that fall outside of some range specified by the scientists.
3. Grid:	Create a uniformly distributed dataset by applying a gridding algorithm. Typically, in order to generate contour lines, the underlying data must be uniformly distributed (i.e., a surface).
4. Contour:	Extract isolines from the uniformly distributed dataset and possibly fill in areas with color associated with a color map
5. Annotate:	This optional task includes adding a legend/color scale to the map. For non colored contour maps (i.e. only isolines) contour labels on the map may suffice. For color maps, color scales may be required.

of toolkit operators and their associated function.

- transforming: converting datasets from one format to another format. Metadata may be lost during transformations if the new format does not support metadata.
- filtering: removing data points (if operating in value state) or removing voxels/pixels (if working in view state) [5]
- gridding: interpolating data points needed to create a grid structure. Many operators that extract isolines and volumes require the input data to be stored as a regular or irregular grid.
- mapping: extracting view geometries such as isolines, surfaces, and volumes from datasets

– rendering: projecting geometries into pixels
– annotating: integrating metadata descriptions or legends into the view
– viewing: presenting rendered artifacts to the screen for viewing or further manipulation (i.e., zoom, crop, rotate)

From the list, mapping, rendering, and viewing are most related to visualization because they are responsible for extracting and processing required views. Other operations such as transforming, filtering, and gridding may be common among scientific processing in general but in the scope of visualization serve primarily as a means for tailoring the input dataset into a form that can be mapped, rendered, and viewed; this distinction is made in [5]. It is up to the user to be able to distinguish between the different operator types and understand which mappers and renderers can generate the desired view. To complicate matters, most toolkits support large number of operators, from 60 supported by Generic Mapping Tools (GMT) [24] to over well over 200 supported in Visualization Toolkit (VTK) [21] and NCAR Command Language (NCL) [12]. Additionally, the different classes of operators are not disjoint; many toolkits support *composite operators*, which are single modules that serve more than one function. GMT for example has operators that perform both gridding and format transformations, while NCL contains operators that both extract geometries (i.e., mapping) and incorporate annotations with legends and toolbars into these geometries. Additionally, viewers like GhostScript can both render geometries specified in PostScript files and present resultant images to the screen for further operations such as zooming.

Finally, toolkits tend to provide sets of operators that serve only a specific visualization goal such as 3d rendering (e.g., VTK) [21], 2d maps (e.g., GMT, NCL) [24, 12], speed-up due to parallel processing (e.g., VTK) [15, 21], graphical user interface (GUI) composable (e.g., IBM Explorer, IRIS, and ParaView) [17, 13, 15], or renderings specific to information (e.g., Visualization Spreadsheet, Titan VTK, Prefuse) [6, 25, 14]. With such diversity of views, toolkits, and operator sets, users may need to expend substantial effort researching each toolkit through reading manuals, trial and error programming, or consultation with a visualization expert.

Once a toolkit is finally selected, users are responsible for writing programs that correctly interface with the toolkit operators. These programs need to orchestrate the sequential execution of operators into sequences known as visualization pipelines. A property of visualization pipelines is that each operator output feeds as input into the next operator specified in the sequence (i.e., unix pipes). Although visualization pipelines can be supported by different toolkits, commonalities between pipelines usually exist:

1. first stages of pipelines are usually responsible for transforming source data into a common data model
2. pipeline stages usually consist of mappers and renderers and although the rendering process itself can be described as a sub pipeline, most toolkits abstract this process into a single operation

3. final pipelines stages may contain transform rendered data into a format that viewers know how to present

Despite similarities among pipelines, these similarities are mainly found at a conceptual level and it may be difficult to realize the pipeline similarities in software because of implementation nuances such as:

- object instantiation code: if the orchestrating language is object oriented, some code is required to manage object resources
- transformation code: if the source dataset cannot be directly ingested by the renderer
- interfacing code: if the target toolkit operators are not accessible through simple function or method calls by the orchestrating language
- argument setting: some arguments of toolkit operators are not simple scalars but rather mathematical functions such as VTK opacity and color functions that control rendering
- granularity of operators: a single operator may implement many different tasks or a single task may be implemented by a set of operators

3.1 Generic Mapping Tools Pipeline

GMT supports a set of operators specific for visualizing 2D datasets (i.e., scalars referenced by 2D spatial/temporal coordinates). A pipeline containing both GMT and GhostScript executables is shown in Figure 3, which is generates gravity map visualizations specified in Section 2. In Figure 3, gravity datasets

Fig. 3. GMT Pipeline that visualizes gravity data as a 2D contour map

Operator	*Associated Parameters*	*Description*
1. GNU awk	columns	filters metadata from XYZ columns
2. nearneighbor.exe	spacing, radius	generates uniform grid in netCDF
3. grdContour.exe	interval, projection	generates contour map
4. ps2pdf.exe	none	converts PostScript to PDF

are filtered by *awk* remove header information related to the gravity dataset. The cleaned point data is then gridded by *nearneighbor.exe*, which generates a uniform point distribution encoded in netCDF [19]. The parameter *grid spacing* refers to spacing between points in the resultant grid while *search radius* specifies the circular area used for collocating points considered in interpolations. *grdContour* extracts isolines from the grid and is configured by *contour interval* that controls the number of generated contours and *projection* that specifies how the contour lines are placed onto a curved surface in 2D space (i.e., the Earth); valid arguments for a projection include Mercator and Lambert. The isoline geometric definitions are encoded in a postscript file, which is transformed into a PDF document by the GhostScript operator *ps2pdf.exe*.

Figure 1 was actually generated by the GMT pipeline and it is important to note that some properties of the map such as the labeling of the axes were generated without being specified. In the case of underspecified arguments, GMT rolls back to using default configurations. Although the *Associated Parameters* column represents only a subset of the actual configurations available for each operator, these parameter dont have any default arguments and must be specified by users.

3.2 NCAR Command Language Pipeline

Like GMT, NCL provides many operators that support the visualization of 2D data, although NCL provides renderings of 3D data as well. Figure 4 presents an NCL pipeline that generates a gravity contour map. The pipeline is very

Fig. 4. NCL Pipeline that visualizes gravity data as a 2D contour map

Operator	*Associated Parameters*	*Description*
1. readAsciiTable	dimensions, delimiter	reads in ascii data
2. csa2	array of coordinates	approximates cubic spline
3. gsn-csm-contour-map	map/annotation configurations	generates contour map
4. ps2pdf.exe	none	converts PostScript to PDF

similar to GMT: the ascii-tabular gravity data must be transformed to a format that can be ingested by the gridding operator *csa2*. Once gridded, isolines can be extracted and the output geometry encoded in PostScript. *gsn-csm-contour-map* operator can be configured by over 30 different argument values to control the contour map rendering and annotations, such as title and color bar orientation and dimensions.

3.3 GMT vs NCL

Disparities in toolkit architectures are not evident in these different pipeline examples. However, if we had added the requirement that a legend must accompany the contour map, we would start to see different architecture philosophies: to modularize or parameterize. GMT provides a relatively cohesive set of operators compared to NCL, which provides more powerful functions that expose highly configurable parameter sets. For example, GMT has separate operators for generating raster images, contour maps, and map annotations such as titles, legends, and color scales. The single NCL operator *gsn-csn-contour-map* supports all these functions and a possible limitation to this flexibility is that users need to understand that although the operator label may indicate that it can generate contour maps, it is actually capable of generating much more. VisKo aims to describe the functionality provided operators to avoid this kind of ambiguity.

4 Visualization Ontology

The Visualization Knowledge (VisKo) ontology defines a canonical visualization language from which to describe different views, functionalities provided by different visualization toolkit operators, and parameter sets. The collections of views, operator, and parameter descriptions comprise our visualization knowledge base, and are used by inferencing agents to automate visualization pipeline composition. The ontology defines different graphical representations of data and information such as isolines, isosurfaces, volumes, networks, and graphs. The ontology also defines the different kinds of operators described in the previous Section 3 and what data they can operate on without defining any execution specific properties. Regarding execution, OWL-S [8] is used to supplement the conceptual operator concepts with execution specific concepts and properties. To distinguish between different visualization concerns, the visualization conceptual space is decomposed and defined by three ontologies: VisKo-View, VisKo-Operator, and VisKo-Service.

The following describes the different VisKo ontologies and presents a UML-like notation describing their structure, where boxes represent concepts, purple arrows indicate "isTypeOf" relationships between concepts, and black arrows indicate relationships between concepts specified by a label.

4.1 VisKo-View

Figure 5 refers to different graphical embodiments of data or information such as contour maps, volumes, charts, and networks. The ontology classifies these views according to whether they are constructed from geometric primitives such as points, lines, polygons, and grids or constructed from higher level graphical constructs such as boxes or other figures and connectors. The distinction between view types is expressed in the ontology by the definition of two view specializations: *GeometricBased* and *LayoutBased*, which have either *hasGeometry* or *hasLayout* properties respectively. The range of the *hasGeometry* property is geometric definitions defined in an ESIP datatype ontology [11]. Future work includes further characterizing views according to the set of attributes they support such as color, opacity, shadowing, orientation, position, and projection [23, 2]. In order to allow users to request for visualizations without having any regard for implementation details such as specific service parameters, we need a link between view attributes and operator parameters that set or configure those view properties. For example, VTK parameters such as *xRotation*, controls the amount a view is rotated about on the X axis, but this parameter can be associated with the more general concept *orientation*.

4.2 Operators

Figure 6 defines different classes of operators: viewers, mappers, and transformers, and declares a common property, *operatorsOn*, which declares what data

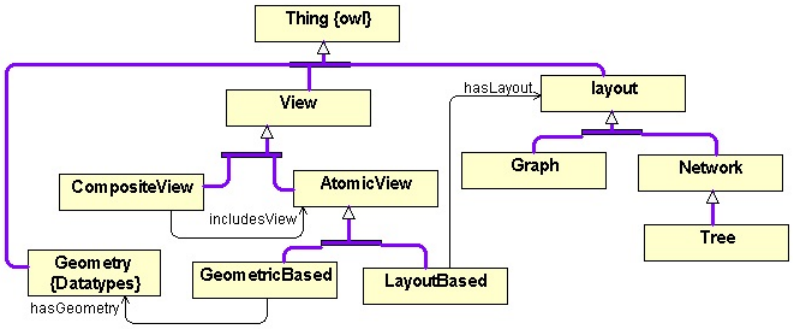

Fig. 5. View Ontology

format an operator processes. *Viewers* are responsible for presenting a generated view and possibly rendering the view onto the screen and include a range of applications such as Adobe PDF Viewer, Ghostscript, and Web browsers. Other, more visualization centric viewers, include applications such as the VTK ParaView [15]. The benefit of specialized 3D viewers such as ParaView is that they support richer interactions by allowing users to rotate, scale, and even change colors and opacity on the fly [7]. However, our current ontology does not define the concept of an interactive viewer.

Mappers, such as *grdContour* identified in Figure 3, are responsible for extracting view geometries from data or formatting information into a layouts such as networks and trees. Mappers are a specialized kind of *Transformer*, because the extracted geometry or layout is usually encoded in specialized format different from the input data format. As the name implies, Transformers such as *ps2pdf.exe* identified in Figure 3 are responsible for simply transforming datasets form one format into another. The *Format* concept is reused from the Proof Markup Language (PML-P) provenance ontology [18].

In VisKo, all operators are either atomic or *composite*, in which case a single operator performs many different functions. This concept is needed especially in the case of toolkits like NCL, in which single operators are responsible for generating a variety of different visualizations and support various annotations such as legends and color scales.

4.3 Services

The operator ontology presented does not define concepts related to execution. To compensate, the operator imports the Ontology Web Language Service (OWL-S) ontology [8], which defines a comprehensive set of concepts and properties needed for describing executable Web services. So long as our visualization operators are exposed as Web services, OWL-S bridges the gap between VisKo-Operator descriptions and their executable counter parts. Thus in order to

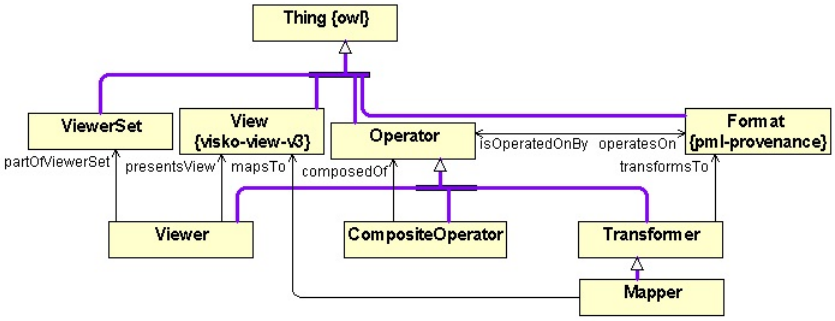

Fig. 6. Operator Ontology

effectively use the VisKo ontology to describe executable pipelines, all operators must have at least one Web service implementation. For example, the operators comprising the pipeline in Figure 3 and 4 would need to be implemented as Web services. While OWL-S ontology defines the notion of a service, the VisKo service ontology presented in Figure 7 extends the OWL-S service concept to include a property *implementsOperator* and thus effectively links operators to services. The separation between operators and service implementations presents an opportunity for a single operator to be implemented by many different services, perhaps supported by different toolkits or services with different performance profiles.

Other concepts defined in Figure 7 include *toolkit*, which represents the different visualization toolkits such as GMT, NCL, and VTK. Services are supported by a particular toolkit and this information is useful if users want to construct pipelines supported by services of only a single toolkit. In general however, hybrid pipelines described in Section 6 are possible. Finally, the service ontology also defines a *Profile*, which is a sort of default configuration for visualizing data or information of a particular semantic type such as: Gravity, Magnetic, Velocity, etc. Profiles contain bindings of parameters with default argument values and are useful when users are unsure of what arguments to assign to service parameters.

4.4 Composition Rules

The VisKo ontologies described are only used to describe the different aspects of a visualization pipeline but do not contain knowledge about how to compose pipelines. VisKo relies on OWL rules to identify possible pipelines, given our knowledge base. Consider our search space as a graph where nodes represent formats and edges represent different operators. In order for an operator edge o to exist between format nodes A and B, edge o must represent an operator that *operatesOn* format A and *trasformsTo* format B. Thus our search graph contains

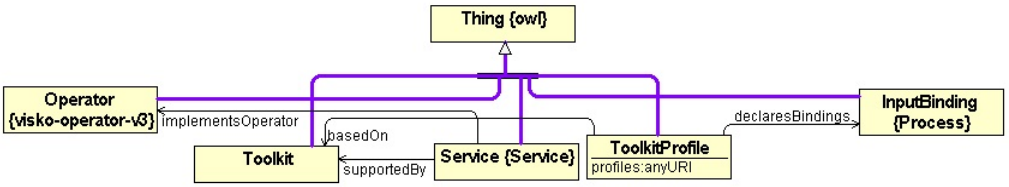

Fig. 7. Service Ontology

all possible combinations of transformations where only a subset of the paths yield meaningful visualizations and the OWL rules make these paths explicit.

Provided we know what view a user wants their data projected as, what viewer they want to use to interact with the view, and what format their current dataset resides in, we can search for pipelines of operators that can generate the view and transform it into a format that the viewer can present. Consider the pseudo code below that identifies such pipelines:

Fig. 8. Search Pipelines Functions

Functions	*description*
mappedBy(view)	returns mapper that generates View
operatesOn(operator)	returns format ingested by operator
transformsTo(operator)	returns format output by operator
depthFirstSearch(KG, sFormat, eFormat)	returns edges (i.e., operators) that link sFormat to eFormat by traversing KG (i.e., knowledge graph) and beginning from sFormat node

- findPipelines(KG, inputFormat, view, viewer)
- mapper ← mappedBy(view)
- targetFormat ← operatesOn(mapper)
- pipeline1 ← depthFirstSearch(KG, inputFormat, targetFormat)
- inputFormat ← transformsTo(mapper)
- targetFormat ← operatesOn(viewer)
- pipeline2 ← depthFirstSearch(KG, inputFormat, targetFormat)
- fullPipeline ← pipeline1 + pipeline2

This simple function *findPipelines* identifies the format operated on by the mapper that generates the required view and performs a depth first search on the knowledge graph to identify a path from the inputFormat node to the targetFormat node. It then repeats the process to identify a path from the format

output by the mapper to the format operated on by the specified viewer. The algorithm can be expressed by OWL rules using property chaining and transitivity, and it is in this form that our algorithm is expressed. The following horn rules are in a more human friendly format than the RDF/XML format they reside in. Inference engines such as Pellet apply the composition rules to our

Table 1. Pipeline Composition Rules

isOperatedOnBy(?Fmt1, ?A) :-	operatesOn(A, Fmt1)
canBeTransformedTo(?Fmt1, ?Fmt2) :-	isOperatedOnBy(Fmt1, A), transformsTo(A, Fmt2).
canBeTransformedTo(?Fmt1, ?Fmt2)	is a transitive rule

knowledge base and infer the pipeline paths. The derivation trace that Pellet used to infer *canBeTransformedTo* clause will tell us the order of operators (i.e., our pipeline) needed to transform our input format into the format required by the view mapper. Note however, that the rules prescribe no method for gathering the augment set that will feed into the operators of the pipeline. VisKo relies on separate methods for determining the appropriate argument set described in Section 5

5 VisKo In Use

5.1 Encoding Gravity Map Visualization Operators

Using VisKo concepts and properties, we can describe toolkit functions such as GMT and NCL. Figure 9 shows statements in our visualization language describing the operators comprising the GMT pipeline in Figure 3.

Fig. 9. GMT Operators Description

Operator	*Subject*	*Predicate*	*Object*
nearneighbor.exe	nearneighbor.exe	is a	transformer
	nearneighbor.exe	operatesOn	tabular-ascii
	nearneighbor.exe	transformsTo	netCDF
grdContour.exe	grdContour.exe	is a	Mapper
	grdContour.exe	mapsTo	isolines
	grdContour.exe	operatesOn	netCDF
	grdCotnour.exe	transformsTo	PostScript
ps2pdf.exe	ps2pdf.exe	is a	Mapper
	ps2pdf.exe	operatesOn	PostScript
	ps2pdf.exe	transformsTo	Portable Document Format

The statements 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 knowledge base that the Pellet reasoner applies our composition rules to.

5.2 Inferring Pipelines

Given the knowledge description in Figure 9, the Pellet reasoner could apply the inverse and transitive property rule in Figure 1 to infer the following statements about the operators. Given this set of inferences, we can easily query

Fig. 10. Inferred Statements

Subject	*Predicate*	*Object*
tabular-ascii	isOperatedOnBy	nearNeighbor.exe
netCDF	isOperatedOnBy	grdContour.exe
PostScript	isOperatedOnBy	ps2pdf.exe
tabular-ascii	canBeTransformedTo	netCDF
tabular-ascii	canBeTransformedTo	PostScript
tabular-ascii	canBeTransformedTo	Portable Document Format
netCDF	canBeTransformedTo	PostScript
netCDF	canBeTransformedTo	Portable Document Format
PostScript	canBeTransformedTo	Portable Document Format

whether a dataset in a given format can be transformed to a format operated on by some mapper or viewer. If in fact a format can be transformed to another target format, we can access sequence of steps used to derive the canBeTransformedTo statements and extract the sequence of operators (i.e. pipeline) that could perform the transformations.

5.3 Pipeline Execution

One a pipeline has been identified and before it is executed, we must configure the operators composing the pipeline with an appropriate argument set. On approach is to simply rely on the default values specified in a *Profile* described in Section 4. Based on the semantic type of the dataset being visualized, a VisKo module will retrieve the corresponding profile and extract the arguments specified in profile. These arguments will then be passed to the operators as they are executed one-by-one. Another approach is to allow the users to explicitly specify these parameter arguments by prompting them as the pipeline is being executed. Or even yet another approach is to let the users *query* for their visualization by providing a set of specifications the visualization must satisfy (described in the next section), including parameter value bindings. Any parameters that were left

unspecified could be bound with the parameter bindings in the profile, similarly to how GMT toolkit defaults to a base configuration.

5.4 Visualization Query

Given that VisKo can assemble pipelines from a knowledge base provided some information such as the requested view and viewer, we might consider the possibility of having users declare for visualizations declaratively. In Structured Query Language (SQL) [16] relational database setting, declarative SQL queries are transformed into query plans (i.e. pipelines of relational algebra operators) that compute or fetch the information requested in the query. With VisKo, we are afforded the same opportunities for such an environment: users can request for visualizations by only declaring the desired view and viewer, along with some information about the input data such as the semantic type and format, and VisKo will synthesize a visualization pipeline that generates the requested view in a format that the viewer can ingest [9]. Users can specify for visualization without specifying any execution specific details such as services or operators.

6 Hybrid Pipelines

Some of the operators composing the two pipelines presented in Section 3 can be interchanged with each other. For example, the GMT operator *nearneighbor.exe* outputs netCDF grids, which can then be read into NCL and used by *gsn-csm-contour-map* to generate a contour map. Similarly, the gridded output of NCL operator *csa2* can be written as a netCDF file and fed into GMT *grdContour.exe* to generate a contour map. The resultant *hybrid* pipelines are results of our canonical descriptions which overcomes the limitations of having the toolkit operators exposed in their source languages. The advantages of hybrid pipelines may have to do with performance; perhaps you prefer the quality of some toolkit's plotting capabilities, but know of another toolkit's interpolation routine that is superior in performance.

7 Related Work

Duke and Brodlie [10] proposed a visualization ontology that was initially sketched in a workshop report [1]. 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 [10] also speak of a separation of concerns between logical and physical layers, where logical layers may refer to our conceptual description of pipeline operators and the physical layer may

correspond to our OWL-S services. If this is the case, then VisKo has found a use for OWL-S to supplementing the Brodlie ontology with execution specific modelling concerns. Another interesting similarity is that neither of our ontologies distinguishes between information and scientific visualization in order to describe the visualization conceptual space.

To the best of our knowledge, Duke and Brodlie have not actually employed their ontology to construct an RDF knowledge base of executable visualization modules. Where the Duke and Brodlie ontology ends at conceptual pipelines, which may not have executable bindings, VisKo continues to describe executable details associated with real visualization toolkits and relies on concepts such as *Toolkit*, *Service* and *Parameter Binding*. Additionally, we are not aware that Duke and Brodlie have defined any rules that help compose pipelines and thus automate the visualization process.

Other authors have proposed models for the purpose of taxonomizing the set of available visualization techniques. Shneiderman proposed a visualization model in terms of two dimensions: task and data type. Given these dimensions, users could look up what kind of visualization would suit a specific task (i.e., browsing or searching) and kind of data you were working with (i.e., 1D, 2D, 3D, or 4D) [22]. This model however did not describe the visualization process itself but provided more of a direct mapping between a kind of data and technique for how it should be rendered. A more process oriented model is Chi's Data State Model [5, 4], 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). Although the data state model can be used to classify visualization techniques [3], there are two systems that actually implement the model, Visualization Spreadsheet [6] and Prefuse [14].

The Data State model classifies operators only as a *tranformers* or *in State* and VisKo embraces the notion of transformers; an operator that progresses data in some state to a state closer to the view. VisKo assumes that as the data state changes so does its format, and this function of transformers is captured in the VisKo ontology. In the future, VisKo might provide a definition for the concept *filter*, a kind of *inState* operator. Because inState operators do not progress data towards the view state, it is difficult to infer when to inject these operations into the pipelines, unless explicitly requested by the user.

There has also been work in defining declarative languages for specifying visualizations. Protovis is one such tool that allows users to associate components of their datasets, such as variables, coordinates, and meta-data with different kinds of graphical *marks* such as bars, dots, and lines. Users of Protovis declare these mappings of data to marks using a language that is defined at a higher level than what is provided by graphical programming languages such as OpenGL, which require users to specify graphics in terms of points, lines, and polygons. However, the Protovis language is still flexible enough for users to specify a large number of customized charts, graphs, and networks. The goal of VisKo queries is the same as Protovis; we want to alleviate user from having to write visualization code and focus primarily on configuring what view they want. The

two systems however define views at different granularities. Protovis users are provided with the capability of specifying custom views whereas VisKo users can only configure existing kinds of views. In fact, VisKo users can only request and configure views generated by mapper operators and so the set of supported views is directly limited by the number of mapper operators described in the knowledge base.

8 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 a set of pipeline composition rules to encoded visualization knowledge. 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 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 the support for this work granted by the CYBER-ShARE Center for Excellence.

References

1. K. W. Brodlie. Visualization Ontologies. http://www.nesc.ac.uk/talks/393/vis_ontology_report.pdf.
2. S. K. Card and J. Mackinlay. The structure of the information visualization design space. In *Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, pages 92–, Washington, DC, USA, 1997. IEEE Computer Society.
3. Ed H. Chi. A taxonomy of visualization techniques using the data state reference model. In *INFOVIS '00: Proceedings of the IEEE Symposium on Information Vizualization 2000*, page 69, Washington, DC, USA, 2000. IEEE Computer Society.
4. Ed H. Chi. Expressiveness of the data flow and data state models in visualization systems. In *AVI '02: Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 375–378, New York, NY, USA, 2002. ACM.
5. Ed Huai-hsin Chi and John Riedl. An operator interaction framework for visualization systems. In *INFOVIS '98: Proceedings of the 1998 IEEE Symposium on Information Visualization*, pages 63–70, Washington, DC, USA, 1998. IEEE Computer Society.
6. Ed Huai-hsin Chi, John Riedl, Phillip Barry, and Joseph Konstan. Principles for information visualization spreadsheets. *IEEE Comput. Graph. Appl.*, 18(4):30–38, 1998.

7. M. C. Chuah and S. F. Roth. On the semantics of interactive visualizations. In *INFOVIS '96: Proceedings of the 1996 IEEE Symposium on Information Visualization (INFOVIS '96)*, page 29, Washington, DC, USA, 1996. IEEE Computer Society.
8. et. al. David Martin, Mark Burstein. OWLS: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/.
9. Nicholas Del Rio and Paulo Pinheiro da Silva. Visualization queries. *Technical Report*, 0, 2010.
10. D. J. Duke, K. W. Brodlie, and D. A. Duce. Building an ontology of visualization. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 598.7–, Washington, DC, USA, 2004. IEEE Computer Society.
11. Earth science information partners (esip) federation datatype ontology. http://wiki.esipfed.org/index.php/Data-Service-Ontologies.
12. National Center for Atomospheric Research (NCAR). NCAR Command Language (NCL) Reference Manual. http://www.ncl.ucar.edu/Document/Manuals/Ref_Manual/.
13. David Foulser. Iris explorer: a framework for investigation. *SIGGRAPH Comput. Graph.*, 29(2):13–16, 1995.
14. Jeffrey Heer. Prefuse: a toolkit for interactive information visualization. In *In CHI 2005: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430. ACM Press, 2005.
15. Alan Heirich, Bruno Raffin, and Luis Paulo Dos Santos. Remote large data visualization in the paraview framework, 2006.
16. International Organization for Standardization (ISO). *SQL Part 2: Foundation (SQL/Foundation)*, 2008.
17. Bruce Lucas, Gregory D. Abram, Nancy S. Collins, David A. Epstein, Donna L. Gresh, and Kevin P. McAuliffe. An architecture for a scientific visualization system. In *VIS 92: Proceedings of the 3rd conference on Visualization 92*, pages 107–114, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
18. Deborah McGuinness, Li Ding, Paulo Pinheiro da Silva, and Cynthia Chang. PML2: A Modular Explanation Interlingua. In *Proceedings of the AAAI 2007 Workshop on Explanation-aware Computing*, Vancouver, British Columbia, Canada, July 22-23 2007.
19. Network common data form netcdf. http://www.unidata.ucar.edu/software/netcdf/.
20. Pan merican center for earth and environmental studies. http://research.utep.edu/Default.aspx?alias=research.utep.edu/paces.
21. Will Schroeder, Kenneth M. Martin, and William E. Lorensen. *The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.
22. Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 336, Washington, DC, USA, 1996. IEEE Computer Society.
23. M. Tory and T. Moller. Rethinking visualization: A high-level taxonomy. pages 151 –158, 2004.
24. P. Wessel and W. H. F. Smith. New, improved version of generic mapping tools released. *EOS Transactions*, 79:579–579, 1998.
25. Brian Wylie and Jeffrey Baumes. A unified toolkit for information and scientific visualization. In *VDA*, page 72430, 2009.

This article was processed using the LaTeX macro package with LLNCS style