skML a Markup Language for Distributed Collaborative Visualization

D.A. Duce and M. Sagar

Department of Computing, Oxford Brookes University, UK

Abstract

This paper describes a reference model and a markup language for representing processing and dataflow in distributed collaborative visualization applications. The language, skML, enables processing to be represented at each of the three layers: conceptual, logical and physical defined in the reference model. skML is an XML application. A particular feature of the approach is the use of RDF to associate annotations with skML elements, for example to describe resource constraints. An interactive editor for skML and prototype tools to use skML with IRIS Explorer and OpenDX are described. Extension to other visualization systems such as AVS and VTK is discussed. The paper concludes by comparing skML to other languages and environments for scientific workflow.

Categories and Subject Descriptors (according to ACM CCS):
H.5.3 [Group and Organization Interfaces]: Collaborative computing I.3.2 [Computer Graphics]: Distributed/network graphics I.3.8 [Computer Graphics]: Applications

1. Introduction

Grid computing allows aggregated computing resources to be harnessed for the solution of problems. Visualization has a crucial role to play in this: the use of visualization in scientific computing and problem solving is well-established; the challenge is to enable visualization systems to integrate seemlessly with other components in order to arrive at solutions. The gViz project (funded by the UK e-Science programme) revisited visualization systems in the light of developments in Grid computing. The overall achievements of the project have been described in earlier papers [BDG*04a]. In this paper we develop one particular theme that has not previously been reported in detail, the design of the skML language and associated tools.

Distributed visualization allocates different processing to different machines; improving performance is often a motivation for this, though there are other motivations, for example security of the primary data source. Collaborative visualization allocates different parts of the human processing to different people, who might be separated geographically. The motivation for this comes from the recognition of the importance of team activity in many endeavours. The team members may well be drawn from different disciplines and bring different expertise to bear on a problem. The visualiza-

tion tools that each uses may well differ, and hence the need to address issues arising from heterogeneity of visualization systems and other resources is fundamental. For a recent review of this area see [BDG*04b].

There is growing interest in the Grid community in scientific workflow languages (see section 8). From a rather different starting point, notably the description of processing networks in modular visualization environments, we have developed an XML application, skML, for describing processing within distributed and collaborative visualization. Thinking about this problem led to the development of a layered reference model for visualization and caused us to think about the need for ontologies for visualization, a topic of current and future work. Tool support for skML has been developed, including tools to interface skML to IRIS Explorer [Wal04] and a proof-of-concept tool for OpenDX [Ope]. Extension to other systems is discussed in section 9.

The gViz reference model is described next. Section 3 describes the skML language itself and the following three sections describe the skML visual editor and support developed for IRIS Explorer and OpenDX. Section 7 discusses how this work prompted the gViz project to begin to think about ontologies for visualization. Related work is discussed in section 8.

2. gViz Reference Model

We regard the visualization process as an ordered sequence of work tasks, where results from one work task are input to a subsequent work task. This notion corresponds closely to the dataflow network models such as the Haber and McNabb reference model [HM90] and the implementation models of modular visualization environments such as IRIS Explorer and AVS. The gViz model extends this idea by recognising three different layers at which it can be applied: a *conceptual layer* where the network is defined in terms of abstract processes independent of any software or physical resources with which it might eventually be realised; a *logical layer* which binds in the software resources; and a *physical layer* which binds in computing resources.

The conceptual layer captures the intention of how application data should be transformed to a visual presentation. Since we are interested in collaborative visualization, this layer also captures something about the collaborative nature of this transformation, and the activity of each participant.

The logical layer binds the conceptual model to a particular configuration of software entites. Each entity might be a component, or a module in a modular visualization sytem, or a function from a procedure library. There is no constraint that conceptual entities of the same kind be bound to logical entities of the same kind. In a collaborative setting, for example, different participants might be using different software to achieve the same result. The logical layer can though introduce constraints on resources, for example, particular processor characteristics, or requirements for co-location to ensure that performance criteria can be met.

The physical layer realises the logical specification in terms of a binding of components to particular physical resources. The model does not assume that this binding is static; it may change as the computation proceeds and resource requirements change.

At each of the layers the model describes the processing as a graph of components and the flow of data and control between them. The model is deliberately avoids defining precisely what is meant by a component, leaving this as an abstract concept that can be grounded as appropriate when the model is applied.

One application of descriptions of this kind is to recognise that a processing description forms a part of the provenance of a visualization, describing how the visualiztion was produced, in such a way that this could be reproduced precisely using the exact resources used originally (description at the physical layer), resources of the same kind (description at the logical layer) or other resources chosen to achieve the same conceptual effect (description at the conceptual layer).

The next section describes skML, which aims to capture the visualization process at any of the three layers.

3. The skML Language

3.1. Basic Features

Two insights are key to the design of skML: firstly that the visualization application can be regarded as a graph whose nodes represent processing and edges represent data flows between processing entities. The second insight is to regard the nodes and edges as resources to which annotations can be attached. An annotation might, for example, describe the type of processing entity the node represents (an IRIS Explorer module, say), or might express constraints such as the kind of processor the node requires.

The design of skML drew inspiration from the Skm scripting language in IRIS Explorer and from general-purpose XML applications for representing graphs, such as XG-MML (eXtensible Graph Markup and Modelling Language) [XGM]. The first version of skML was very similar to Skm, but the two diverged somewhat as the gViz reference model, described in the previous section, evolved and skML began to be used in a more general way. We borrow some terminology from IRIS Explorer. A processing graph will be called a *map*, and the basic components of maps are *modules* connected by *links*. Modules may have associated *parameters*. The skML language has a simple structure. The main elements are `skml`, `map`, `module`, `param` and `link`. The `skml` element is the root element of a skML document. The `map` element describes a processing graph of modules and links. The element's attributes include:

id a unique identifier for the map.
style used in the editing tool to distinguish one map from another and to locate the map on the editor window.

The `module` element can be used to express the creation of a new module instance, or destruction or modification of an existing module instance. The attributes of this element include:

id a unique identifier for the element. This is used to identify an instance of a module within a map and hence has to be unique across the collection of skML files related to that map. It is represented by the XML ID type.
name the module's name. skML uses name to mean the type of module the element represents, for example, a module to compute isosurfaces.
ref a reference to a module (a value defined in an `id` attribute).
style used to position the graphical representation of a module on an editor window.
action action to be taken when interpreting the element. Values include: `create`, `destroy`, `modify` and `create-modify`. The default value is `create-modify`.
in-port name of input port.
out-port name of output port.

A simple example is shown below. These elements denote

the creation of modules of types `ReadImg` and `Display-Img` respectively.

```
<skml>
 <map id="DisplayImage">
  <module id="RImg" name="ReadImg" />
  <module id="DImg" name="DisplayImg" />
 </map>
</skml>
```

We developed a visual editor for skML (see section 4) and in order to meet a requirement to save and load skML files with the same visual appearance, `module` elements can also be given a `style` attribute which can be used to record the position of the module's representation on the display surface. The syntax follows the CSS style syntax. An example is shown below which adds a style attribute to the previous module instance to record the position of the top left hand corner of the representation.

```
<module ref="RImg"
  style="Left:300;Top:100;" />
```

The `param` element is used to set a parameter value, or value range. The value is given as the content of the element. The attributes are:

name the name of the parameter.
min the minimum value of the parameter.
max the maximum value of the parameter.

A simple example is shown below. The markup sets a value for the `filename` parameter of the module with `id` attribute `RImg`.

```
<module ref="RImg">
  <param name="filename">pic.jpg</param>
</module>
```

The `link` element can be used to connect or disconnect two modules. The main attributes are:

id a unique identifier for the link element.
ref a reference to a `link` element. This would be used, for example, to identify a link that is to be disconnected.
action whether the element denotes the creation (`connect`) or destruction (`disconnect`) of a link. The default value is `connect`.

A simple example is shown below.

```
<link id="con1">
  <module ref="RImg" out-port="Output" />
  <module ref="DImg" in-port="ImgIn" />
</link>
```

The `link` element describes the connection between the two modules, the `out-port` of the ReadImg module is connected to the ImgIn port of the DisplayImg module. Each type of module will have its own set of available port names. A link is thus defined by the ports at the ends of the link. Because in general we wish to describe processing graphs in

which the modules can be drawn from heterogeneous software systems, we do not impose any *a priori* typing constraints on links, modules and ports.

Disconnecting the link defined above is described by the following skML.

```
<map id="DisconnectDisplay">
  <link ref="con1" action="disconnect" />
</map>
```

Having the ability to describe disconnection of links and destruction of modules gives skML the flavour of an audit trail. Visualization maps are often changed during the course of a collaborative visualization activity and it is useful to be able to record this in skML.

An XML Schema has been developed to define the skML language.

3.2. skML annotation

The skML language makes no assumptions about the meaning of module names, ports, parameters etc. Thus skML is able to represent maps in which the modules can be anything. All we know about a module is that it has a name, a distinguishable input port and output port, and can be parameterised. We use the Resource Description Framework [RDF04] to provide additional information about the entities in a skML document. RDF provides markup for describing resources; one way to think of this is that the RDF descriptions annotate the skML document. An example is shown below.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/
    02/22-rdf-syntax-ns#"
  xmlns:v="http://www.gviz.org/skML/">
 <rdf:Description about="DImg">
  <v:Type>IRISExplorer</v:Type>
 </rdf:Description>
</rdf:RDF>
```

This annotation is saying that the module with `id` attribute `DImg` is an IRIS Explorer module.

We exploit the fact that RDF provides mechanisms for associating descriptions with resources but does not define what those descriptions contain. The contents of the descriptions are defined in other namespaces. In the example above we used the namespace prefix "`v`" to identify the markup language in which types of modules are defined. RDF thus provides a flexible and extensible mechanism for associating descriptions of many different kinds with skML entities.

Other kinds of annotation can also be described in this way, for example, annotation that describes the allocation of modules to physical resources at the physical layer or constraints on resource allocation at the logical layer.

Placements of the RImg and DImg modules in the example above could be described by the annotations shown below.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/
  02/22-rdf-syntax-ns\#"
  xmlns:v="http://www.gviz.org/skML/">
  <rdf:Description about="RImg">
    <v:Type>IRISExplorer</v:Type>
    <v:PhysicalLocation
      rdf:resource="http://cms.brookes.ac.uk/
        pc1"/>
    </rdf:Description>
  <rdf:Description about="DImg">
    <v:Type>IRISExplorer</v:Type>
    <v:PhysicalLocation
      rdf:resource="http://cms.brookes.ac.uk/
        pc2"/>
  </rdf:Description>
</rdf:RDF>
```

In the example the physical resources to which modules are allocated are named with URIs and the names are attached to the modules as anotations. We have used URIs in the http scheme to denote processor resources for illustration. Proper application of this idea would give more careful consideration to how resources are named, but this would not affect the general style of the markup.

It was not the aim of the gViz project to develop vocabularies and markup languages for describing computational resources. Other projects are addressing this question, for example the RSL resource description language from Globus [RSL] and the Glue-Schema [Glu] (a conceptual data model for Grid resource discovery and monitoring) in the InterGrid project. An example of a resource constraint that might be attached to a module at the logical level is shown below. The constraint language is a very simple language developed by us to illustrate the general idea.

```
<constraint
    xmlns="http://www.examples.com/specML"
    xmlns:rsl="http://www.globus.com/RSL">
 <and>
  <rsl:executable> a.out</rsl:executable>
  <or> <and>
    <rsl:count>5</rsl:count >
    <rsl:memory operator=">=">64</rsl:memory>
   </and> <and>
    <rsl:count>10</rsl:count >
    <rsl:memory operator=">=">32</rsl:memory>
   </and>
  </or>
 </and>
</constraint>
```

In summary by regarding type descriptions, and performance and other constraints as annotations to be expressed in the RDF framework, we arrive at an extensible approach that shows promise.

3.3. Distributed Collaborative Visualization

The aim of distributed collaborative visualization is to harness the processing power of many humans and many com-puers, each making their individual contribution to some joint endeavour. The approach introduced in the previous section enables us to describe distributed visualization, i.e. maps in which modules are allocated to different processing resources. This can be done at the logical layer where the resources are not explicitly identified, but resource characteristics (e.g. a processor of a particular kind) are expressed as constraints. Using the same annotation mechanism, allocation to actual named resources at the physical layer can be expressed.

To capture the collaborative nature of a visualization map we observe that different parts of the map will be associated with different participants. This leads to the idea that different participants may play different *roles* in a collaboration, and hence we can associate parts of the overall map with particular roles. There may be constraints on the number of participants who can take a given role, for example in a teacher-student setting, the number of students may be unlimited but only one teacher is permitted. Introducing a new participant in a particular role is then just a case of instantiating a particular collection of modules and links and likewise removing a participant involves deleting modules and links.

Roles are represented by different maps within the same document; the id attribute name can be chosen to identify the role. For example a teacher role could be described by the following map which allows the participant to select an image to display.

```
<map id="Teacher" >
  <link>
    <module id="RImg" name ="ReadImg"
      out-port="Output"/>
    <module id="DImg" name ="DisplayImg"
      in-port="Img In" />
    </link>
</map>
```

A student role could be described by:

```
<map id="Student" >
  <link>
    <module ref="RImg" out-port="Output"/>
    <module id="SDImg" name ="DisplayImg"
      in-port="Img In" />
    </link>
</map>
```

This attaches another DisplayImg to the output port of the ReadImg module.

4. The skML Visual Editor

4.1. Functionality

A visual editor for skML has been written. The user interface is based on the Map editor in IRIS Explorer. The basic interface is shown in Figure 1. The left hand pane contains an expandable tree representation of stored map files and lists of available modules. These are organised hieararchically.

No other structure is imposed on this area. The example in the figure shows areas for IRIS Explorer and OpenDX for illustration.
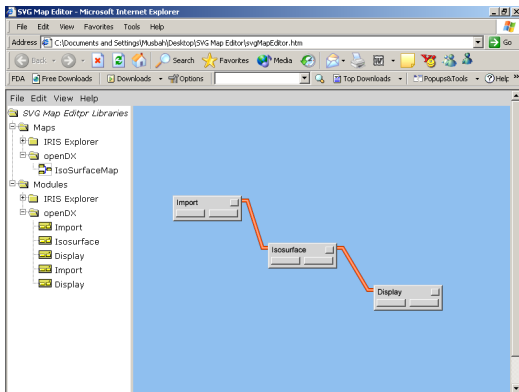

Figure 1: *SVG Map Editor user interface.*

The main functionality provided by the editor is:

1. create/modify maps;
2. import/export skML documents;
3. organize/add new modules to the module palette;
4. introduce new hosts.

The skML editor makes a minimal set of assumptions about modules (that they have (more than one) named input and output port), thus it is not tied to any particular visualization system. The dialogue box for creating a new module instance is shown in figure 2. The application type defines the namespace from which the module is taken.
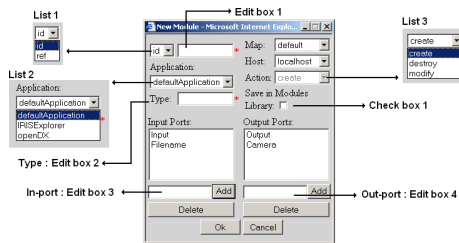

Figure 2: *Create new module type dialogue box.*

If the instance is the first of a new type of module, the type description can be saved in a library.

Some aspects of the configuration of skML are controlled by configuration files (XML documents). The types of application are specified by application elements within a type element. It is thus easy to introduce new types of application.

```
<type>
 <application name="defaultApplication"/>
 <application name="IRISExplorer"/>
 <application name="openDX"/>
</type>
```

Individual modules are also described by XML documents, for example:

```
<module name="Render">
  <inPort name="Input"/>
  <inPort name="Input Camera"/>
  <outPort name="Output Camera"/>
  <outPort name="Snapshot"/>
</module>
```

The module description lists the module's input and output ports. In this example the visualization application to which the module belongs is not expressed in the module description. This could be added as an annotation to the description, or, as we have done in the prototype editor, it can be inferred from the location of this document in the module library hierarchy.

As explained earlier, a skML document can contain more than one map. New maps can be created, or existing maps modified, through the "New/Modify Map(s)" entry in the file menu. Selecting this menu item brings up the dialogue box shown in figure 3.
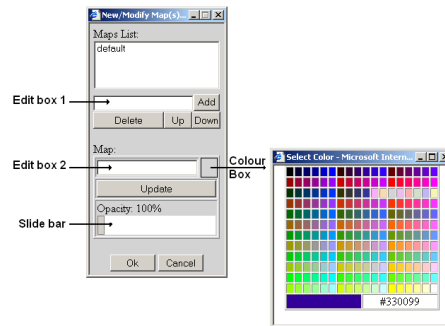

Figure 3: *Maps list dialogue box.*

Edit box 1 and the Add button are used to insert a new map into the maps list. The Delete button deletes a map. The Up and Down buttons are used to change the order in which the maps will appear in the skML document saved from the editor. The properties of a map can be changed by first highlighting the map in the maps list, then using edit box 2 to change the name of the map, or the colour box to change the colour of the map or the slide bar to change the opacity of the map. Opacity was found to be a useful way to highlight particular maps within the collection of maps.

Modules and links have similar graphical representations to those used in IRIS Explorer. Figure 4 shows the tip boxes that provide extra information about modules and links. When the mouse pointer moves over a link, the pop-up box gives information about the names of the input and output ports which the link connects. When the mouse pointer is over the square button at the top right hand corner of the module box, a tip box appears which shows the value of the

module's `id` attribute, the map with which the module is associated and (if appropriate) the name of the host to which it is associated.
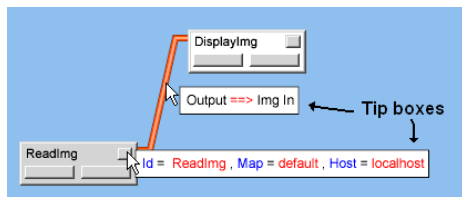

Figure 4: *Information provided by pop-up tip boxes.*

Figure 5 illustrates the steps in creating a new link:

1. bring up the pop-up menu for the first module (by right clicking on the button-shaped rectangle at the top right-hand corner of the module box), select an out-port from the menu (label 1 in figure);
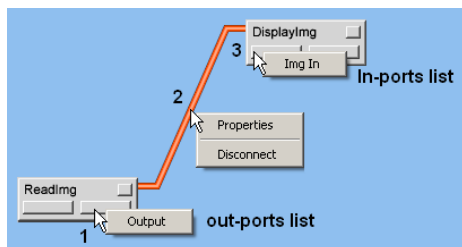2. select an in-port from the pop-up menu for the second module (label 3 in the figure);

Figure 5: *Creating a new link.*

Right-clicking on a link brings up a pop-up menu with options to delete the link or change its properties. Selecting the Properties option brings up a link properties dialogue box in which the map to which the link belongs can be set along with the type of the link (i.e. the type of information the link can carry). We do not preclude specifying maps in which the types of the input and output ports joined by a link are different and hence type information is regarded as annotation of the link. The benefit of this lax approach is that type conversion can be inserted as necessary when the link is instantiated. The skML editor does not itself perform type checking.

4.2. Implementation

The SVG Map Editor runs in a Web browser and is implemented using Scalable Vector Graphics (SVG) for the graphical presentation and JavaScript. Graphical presentation is generated by scripting the SVG Document Object Model (DOM). Two JavaScript libraries were developed: a graphics library to support the creation of graphics in SVG using DOM methods and a user interface library providing similar

functionality to the Java Swing classes. An early version of the libraries was described by Sagar [Sag03]. This approach has proved to be effective in practice.

5. The IRIS Explorer Interface

An IRIS Explorer module (skMLCollaborative) has been written that will taken a skML document and launch a selected map within the document. A Skm script to save IRIS Explorer maps in skML and a stand-alone tool to convert skML documents to IRIS Explorer Skm scripts have also been written.

The COVISA toolkit [WWB97] provides support for collaborative visualization in IRIS Explorer. The skMLCollaborative module automatically generates instances of COVISA modules that are needed in order to transport data from the instance of IRIS Explorer run by one participant to that run by another. Thus many of the details of COVISA are hidden from the author of the skML description of the overall visualization process.

6. OpenDX

As a simple proof-of-concept exercise, a translator from skML to OpenDX [Ope] has been written. A skML map to read a data file, generate an isosurface from it and display the result is shown below.

```
<skml>
 <map id="openDXIsoMap" style="left:147;top:87;
    color:#d4d4d4">
  <link>
   <module id="data" name="Import"
    out-port="out" style="left:0;top:0;">
    <param name="Filename">
     /usr/data/image/watermolecule
    </param>
   </module>
   <module id="iso" name="Isosurface"
    in-port="data" style="left:147;top:73;"/>
  </link>
  <link>
   <module ref="iso" out-port="data" />
   <module id="img" name="Display"
    in-port="in" style="left:310;top:138;"/>
  </link>
 </map>
</skml>
```

The skML file is then translated to OpenDX; we have written a simple utility program for this purpose. For the example above, the translator generates the OpenDX script below. The result of running this script is shown in figure 6.

```
data = Import("watermolecule");
iso = Isosurface(data);
camera = AutoCamera(iso);
img = Display(iso,camera);
```
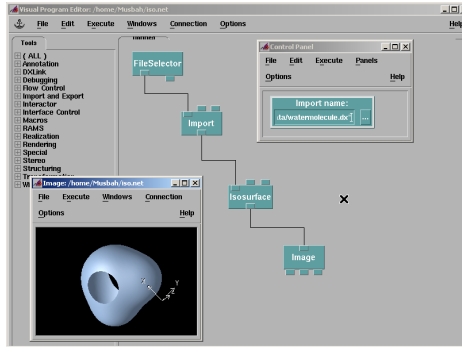
Figure 6: *Script run in OpenDX.*

7. Conceptual level skML

A skML document at the conceptual level can in principle be transformed automatically to corresponding documents at the logical level for *different* software systems. Simple proof of concept translations have been developed for IRIS Explorer and OpenDX to illustrate the idea. In the general case, however, this is far from straightforward as there are no guarantees that modules in one visualization system have counterparts (either one-to-one or through some subgraph of modules) in another, nor are there guarantees that even if the functionality provided by corresponding modules in two systems is similar the semantics correspond. Modules to compute isosurfaces in different systems might well use different algorithms and have different numerical accuracy/stability. Considerations such as these then lead to the question of how one can assign meaning to visualization modules.

One of the approaches to this question is through the development of an *ontology* of visualization. In an ontology concepts are described using a set of constructors with a pre-agreed meaning; for example through a set of relationships that can be asserted between primitives. Since there are fixed ways to define new concepts, ontologies can be made machine processable and this is one of the current research directions in the Semantic Web with technology such as OWL. Given the existence of an ontology for visualization, annotations of modules in skML maps could be used to ground the meaning of modules by referencing concepts defined in the ontology. This is a direction for future work [DBDHss].

8. Related Work

One way of thinking about skML is that it is a language for expressing visualization workflow. There is growing interest in the Grid community in languages for expressing scientific workflows, Kepler, Triana, ICENI and OGSA-DAI are good examples. Each of these projects has developed a workflow language, but from different starting points.

Kepler is a scientific workflow management system

[ABJ*04], providing a GUI to support the creation of complex workflows and a modular programming environment. Workflows can be serialized in an XML application called Modelling Markup Language (MoML) which is designed to allow the specification of parameterized, hierarchically structured collections of components. Processing steps are called *actors* performing computations such as signal processing and statistical operations. Actors have input and output ports that can be linked into a directed graph to allow dataflow between actors; all the systems considered here, including skML, share this fundamental model. Actors can include Web and Grid services and hence support for distributed computation is provided. Interestingly the paper cited above points out that "the workflow also provides the provenance for derived data products, allowing researchers to return to previous states of the data as needed". We also point to the use of skML maps as annotations of data sets and visualizations as a way to capture the provenance of the visualization, the way in which it was generated. In the context of the gViz 3-tier model, this can be done at each of the three levels to capture the higher level intentions of the visualization as well as the precise way in which these were realized.

Triana [ST04] is a graphical Problem Solving Environment for scientific applications, originally developed for use by data analysis scientists in a project concerned with the detection of gravitational waves. It has now grown into a sophisticated environment for supporting distributed workflows that map to Triana Web and P2P services. It also provides a simple XML workflow language for composing components. Components are Java classes with a name, input and output ports and parameters. Code wrapping is used to enable components written in other languages to be used.

The Taverna [Tav] system and Scufl workflow language have evolved in the bioinformatics domain, typically to support workflows for *in silico* experiments expressed as complex chains of database searches and analytical tools. The Open Grid Services Architecture – Data Access and Integration (OGSA-DAI) project has developed middleware to assist with access and integration of data from different sources. It is built around an activity-based workflow where activities can include data retrieval and transformation.

ICENI [MYA*04] (Imperial College e-Science Networked Infrastructure) provides a component programming model to aid developers in constructing Grid applications, and an execution infrastructure. The ICENI component framework has 3 layers, and the separation of concerns has parallels with the gViz 3-tier model. The three concerns are meaning, behaviour and implementation. These concerns are captured as metadata (expressed in XML) as components are constructed. Meaning captures semantic constraints, behaviour captures control and data flow and thread data, implementation captures performance characteristics and platform specific requirements. ICENI too has an XML based language to describe workflows [MMG*02].

We would not argue for superiority of skML as a workflow language over any of the other languages mentioned here; indeed it is probably true to say that in many cases it would be straightforward to translate documents from one language to any of the others. skML differs from these other languages in recognising some kinds of metadata as annotations that can be expressed in RDF.

9. Conclusions

We have described the gViz reference model and the skML language. Tool support for the language has also been described. skML offers a system-independent description of a traditional dataflow network. The work has provided motivation for studying ontologies of visualization and this direction is now being pursued.

The implementation work was based on IRIS Explorer and OpenDX. This raises the question of whether the approach could also be applied to other systems, e.g. AVS and VTK. AVS products have a similar architecture to IRIS Explorer. A scripting language, CLI, is provided and users can write new modules. AVS has also been extended to collaborative working in a similar manner to IRIS Explorer and there would seem to be no conceptual difficulties in developing modules that mirror the IRIS Explorer skML import and export modules. At first sight, VTK is rather different, being a class library. However the VTK model is also a data flow model and the class libraries include process objects (classified into source, filters and mappers), and data objects. Although VTK does not have an equivalent to the module editor in IRIS Explorer and AVS, the toolkit is interfaced to scripting languages including Tcl/Tk and Python. From an initial inspection it would seem possible to translate from skML into a script that would construct the corresponding VTK pipeline, but this has not been confirmed.

Acknowledgements

References

[ABJ*04] ALTINTAS I., BERKLEY C., JAEGER E., JONES M., LUDASCHER B., MOCK S.: Kepler: Towards a Grid-Enabled System for Scientific Workflows. In *Workflow in Grid Systems Workshop in GGF10 - The Tenth Global Grid Forum, Berlin, Germany* (March 2004).

[BDG*04a] BRODLIE K., DUCE D., GALLOP J., SAGAR M., WALTON J., WOOD J.: Visualization in Grid Computing Environments. In *IEEE Visualization 2004* (2004), pp. 155–162.

[BDG*04b] BRODLIE K., DUCE D., GALLOP J., WALTON J., WOOD J.: Distributed and Collaborative Visualization. *Computer Graphics Forum 23*, 2 (2004), 223–251.

[DBDHss] DUKE D., BRODLIE K., DUCE D., HERMAN I.: Do You See What I Mean? *IEEE Computer Graphics and Applications* (2005, in press).

[Glu] Glue Schema. http://www.cnaf.infn.it/~sergio/datatag/glue/.

[HM90] HABER R. B., MCNABB D. A.: Visualization Idioms: A Conceptual Model for Scientific Visualization Systems. In *Visualization In Scientific Computing* (1990), Shriver B., Neilson G.,, Rosenblum L., (Eds.), IEEE Computer Society Press, pp. 74–93.

[MMG*02] MAYER A., MCGOUGH S., GULAMALI M., YOUNG L., STANTON J., NEWHOUSE S., DARLINGTON J.: Meaning and Behaviour in Grid Oriented Components. In *3rd International Workshop on Grid Computing, Grid 2002, volume 2536 of Lecture Notes in Computer Science* (2002).

[MYA*04] MCGOUGH S., YOUNG L., AFZAL A., NEWHOUSE S., DARLINGTON J.: Workflow Enactment in ICENI. In *Proceedings of the UK e-Science All Hands Meeting, ISBN 1-904425-21-6* (2004).

[Ope] OpenDX. http://www.opendx.org/.

[RDF04] Resource Description Framework (RDF). http://www.w3.org/RDF/, 2004.

[RSL] Resource Specification Language. http://www-fp.globus.org/gram/rsl_spec1.html.

[Sag03] SAGAR M.: An SVG browser for XML languages. In *Proceedings of Theory and Practice of Computer Graphics 2003* (2003), pp. 42 – 48.

[ST04] SHIELDS M., TAYLOR I.: Programming Scientific and Distributed Workflow with Triana Services. In *Workflow in Grid Systems Workshop in GGF10 - The Tenth Global Grid Forum, Berlin, Germany* (March 2004).

[Tav] Taverna. http://taverna.sourceforge.net/.

[Wal04] WALTON J. P. R. B.: NAG's IRIS Explorer. In *Visualization Handbook* (2004), Johnson C. R., Hansen C. D., (Eds.), Academic Press. Available from http://www.nag.co.uk/doc/TechRep/Pdf/tr2_03.pdf.

[WWB97] WOOD J. D., WRIGHT H., BRODLIE K. W.: Collaborative Visualization. In *Proceedings of IEEE Visualization '97* (1997), Yagel R., Hagen H., (Eds.), pp. 253–259.

[XGM] XGMML (eXtensible Graph Markup and Modeling Language). http://www.cs.rpi.edu/~puninj/XGMML/.