Automatic Transformation from Geospatial Conceptual Workflow to Executable Workflow Using GRASS GIS Command Line Modules in Kepler[1]

Jianting Zhang, Deana D. Pennington, William K. Michener

LTER Network Office, the University of New Mexico,
MSC 03 2020, 1 University of New Mexico
Albuquerque, NM, 87131, USA
jzhang@lternet.edu

Abstract. Many geospatial models are developed using command line modules of GIS packages. To utilize scientific workflow technology in geospatial modeling, it is important to support command line GIS modules in scientific workflow systems. However, straightforward representation of command line modules as workflow components conflicts with conventional conceptual design patterns. We propose a two-step geospatial scientific workflow composition approach. Simple conceptual workflows are composed in the first step. These allow data type-based workflow validation. The validated conceptual workflows are then transformed automatically into executable workflows using command line modules in the second step. We describe the preliminary implementation of the proposed approach in the Kepler scientific workflow system and demonstrate its feasibility using an example.

1. Introduction

Traditional geospatial data processing functions are provided in the form of command line modules. They take a set of control options, a set of input/output file names, transform input files into output files and optionally output resulting messages to standard output devices (e.g. screen). The Geographic Resources Analysis Support System (GRASS) is the most widely used open source Geographical Information System (GIS) package. Originally developed by U.S. Army Construction Engineering Research Laboratories in 1982, GRASS has evolved into a large system that consists of more than two hundred command line modules ranging from 2D vector/raster data analysis to 3D visualization and image processing [1].

Scientific Workflow technologies have attracted considerable research and application interests during the past years under the framework of grid computing. In a scientific workflow, data are passed as tokens through component ports, which must be wired for the specific data type expected. Ports can also be used to pass additional

[1] This work is supported in part by DARPA grant # N00017-03-1-0090 and NSF grant ITR #0225665 SEEK

specifications required for a particular processing unit, as is the case with components based on command line modules. Configure of the additional ports is based on the syntactic requirements of the command line modules. Therefore, workflow composition from command line GIS modules can be partially automated by making use of the embedded syntax to handle the additional details, freeing the application developer (domain specialist) to focus on the more conceptual aspects of the workflow.

The most straightforward representation of command line modules as workflow components is to treat the control options and the input/output file names as the inputs of a workflow processing unit. The outputs can be text messages and/or an exit code. However, the conventional conceptual design pattern, i.e., inputs-processing-outputs, is greatly encumbered by representing this subsidiary information along with the high-level information during workflow design. Not only do the control options and file names clutter the conceptual workflow, the representation is not intuitive for a domain user. For example, the output data of a conceptual processing unit is treated as an output but the file name associated with the output data in the corresponding executable processing unit is treated as an input. In addition, it is very difficult to perform semantic validation on workflows that use command line modules based on the compatibility of the input/output data types of the workflow components, because all the input file names are string data type and all the exit codes are integer data type.

We propose a two-step approach to compositing geospatial processing workflows: the first step composes a conceptual workflow and the conceptual workflow is transformed automatically into an executable workflow using the embedded information in the command line modules in the second step. The conceptual workflow is closer to users' perception of geospatial processing, which uses semantic geospatial data types (such as vector/raster/tin). The processing units in the conceptual workflows are abstractions of command line modules. Semantic validations can be performed on the conceptual workflows. Only the validated conceptual workflows are allowed to be transformed to executable workflows. The processing units in the executable workflows correspond to the command line modules at the syntactic level. When the executable workflow is executed, the command line options and input/output file names are obtained automatically from the workflow processing units. They are fed to GRASS GIS command line modules that can be either invoked directly within the processing units or output scripts for future execution. While the implementations of the proposed approach are specific for the Kepler scientific workflow system ([2]), we believe the approach is applicable to other scientific workflow systems as well. The proposed approach is motivated by the work in [3] and specifically aims at using command line modules in scientific workflow systems.

The rest of the paper is arranged as follows. Section 2 introduces the Kepler scientific workflow system. Section 3 describes the mapping rules between the command line parameters of GRASS GIS modules and the executable processing unit configurations. Section 4 presents the proposed automatic transformation approach and provides technical details of the implementations. Section 5 is the demonstration and finally section 6 is the summary and future work directions.

2. Kepler Scientific Workflow System

Kepler [2] builds upon the mature, dataflow-oriented Ptolemy II system (Ptolemy [4]). Ptolemy controls the execution of a workflow via so-called directors that represent models of computation. Individual workflow steps are implemented as reusable actors that can represent data sources, sinks, data transformers, analytical steps, or arbitrary computational steps. An actor can have multiple input and output ports, through which streams of data tokens flow. Additionally, actors may have parameters to define specific behavior. An illustration is shown in Fig. 1. Note that Parameter Port is an extension of regular IO Port and its value can either be preset using an associated parameter or updated by the connecting port dynamically as a regular IO Port. Kepler inherits and extends these advanced features from Ptolemy and adds several new features for scientific workflows, such as ontology-based data and actor searching, semantic type checking and advanced object management for distributed execution of workflows.

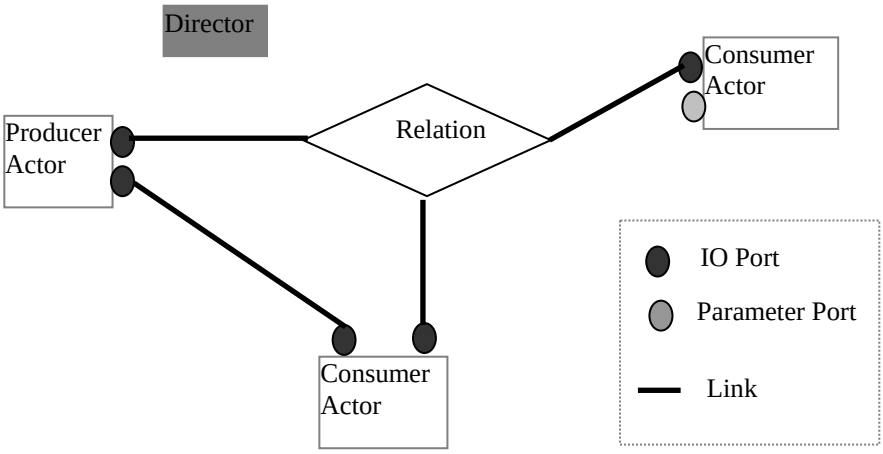

Fig. 1 Illustration of Basic Components in Kepler Scientific Workflow System

Kepler provides an annotated library of workflow components (directors, actors, parameters, etc.). When users drag and drop them into the workflow composition canvas, the data associated with the workflow components are added to the workflow model. Actors and the ports associated with the actors are rendered graphically. Users can then connect two ports or a port and a relation (or creating a link in Ptolemy terminology) by dragging and dropping as well. The workflow composition canvas allows typical types of zooming (in/out/fit) and automatic layout. The automatic layout in conjunction with manual adjustments can produce nice visualizations of workflows. In addition, Kepler provides a panner (or a miniature map) to navigate users through complex workflows in a convenient manor.

Workflow components in Kepler/Ptolemy can be specified in XML using Ptolemy's Model Markup Language (MoML, [4]). Ports and parameters of actors can be added dynamically by modifying the actors' MoML elements. This feature allows create actors to represent command line modules in a GIS package using a single

place-holder actor without actual programming. The composed workflows are internally represented as MoML documents as well. Fig. 2 shows a fraction of MoML document that adds a parameter (property) called “comments” and two IO ports (input_dt and output_dt). It uses a placeholder actor called “util.ConceptActor” to instantiate a geospatial processing actor called “r_buffer” that corresponds to “r.buffer” module in GRASS GIS package. Note that the “.” symbol in GRASS module naming convention is replaced with “_” in MoML because it is reversed for denoting naming hierarchy in Ptolemy.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
   "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<group>

<entity name="r_buffer" class="util.ConceptActor">
        <property name="comments" class="ptolemy.data.expr.StringParameter" value="Creates a raster
map layer showing buffer zones surrounding cells that contain non-NULL category values.">
                <property name="_style" class="ptolemy.actor.gui.style.NotEditableLineStyle"/>
        </property>
        <port name="input_dt" class="ptolemy.actor.TypedIOPort">
                <property name="input"/>
                <property name="type" class="ptolemy.actor.TypeAttribute" value="string"/>

        </port>
        <port name="output_dt" class="ptolemy.actor.TypedIOPort">
                <property name="output"/>
                <property name="type" class="ptolemy.actor.TypeAttribute" value="string"/>
        </port>
</entity>

</group>
```

Fig. 2 MoML representation of conceptual actor corresponding to r.buffer

3. Preparing Actor Libraries

We have developed two separate workflow component libraries. The conceptual workflow component library includes actors necessary for conceptual workflow composition termed as “conceptual actors”. They are jointly developed by Kepler workflow experts and GRASS GIS experts. Basically these conceptual actors are the abstractions of GRASS GIS modules focusing on input and output data types. Controls and optional parameters associated with the GRASS GIS modules are ignored in the conceptual actors for simplicity.

The executable workflow component library includes actors syntactically corresponding to GRASS GIS modules. Similarly they are termed as “executable actors”. The mapping rules are as follows:

1. Each GRASS GIS module is mapped to an “Entity” in the Kepler executable actor library. The name of the entity is the same as the GRASS GIS module and the same as the corresponding conceptual actor.

2. Each mandatory GRASS GIS parameter is mapped to a “TypedIOPort” in the library. If the parameter is a file name, then “_fn” is appended to the name of the port.
3. Each optional GRASS GIS parameter is mapped to a “PortParameter” which can serve both as a port and a parameter in Kepler. If the associated port of PortParameter is connected to an output port, then its value is updated by the output port dynamically. Otherwise the value of the PortParameter is given at the time the actor is created or the parameter is modified.
4. Each control option (“such as “-a”, “-l”) is mapped to a “Parameter” with a “CheckBoxStyle”. The value of the parameter will be either true (the option is set) or false (otherwise).
5. Two additional “StringParameters” are added to each executable actor whose contents are retrieved from GRASS GIS manual. Parameter “comments” describes the functionality of the module. Parameter “commandline” describes the command line syntax of the module. These two parameters are non-functional and for illustration and tutorial purposes only. Thus they are set to “NotEditableLineStyle” indicating read-only.
6. For the parameters in GRASS GIS modules that require choosing one among a limited number of candidates, mandatory or optional, “ChoiceStyle” is used for the corresponding “TypedIOPort” or “PortParameter” and the candidates are used as the properties of the style. The properties are rendered as the items in a dropdown list in Kepler GUI.
7. For the parameters in GRASS GIS modules that require choosing multiple items among a limited number of candidates, mandatory or optional, all the candidates are concatenated as a string. The leading “{“and ending “}” symbols are added to the string to indicate the multiplicity. A property called “_stringMode” is added to the corresponding “TypedIOPort” or “PortParameter” to avoid being evaluated as an expression. This walk-around approach is mostly because Kepler GUI currently does not support multiple choices from a dropdown list.
8. For the parameters in GRASS GIS modules that require multiple numeric (integer or real) numbers, the default values are concatenated to strings using comma as the separation symbol. Thus the ports are essentially string data type and require adding property “_stringMode”. The leading “{“and ending “}” symbols are added as well to indicate the multiplicity nature of the parameters similar to rule 7.
9. An input port (called “input_trigger”) and an output port (called “output_trigger”) are added to each executable actor, both are boolean type. The “output_trigger” is used to send a successful/failure token to the “input_trigger” port of the connecting actor. The links between “output_trigger” ports and “input_trigger” ports define the topology of a workflow.

The executable actor that corresponds to the r_buffer conceptual actor in Fig. 2 is shown in Fig. 3. The graphical representations of the executable actor are shown in Fig. 4.

```
    <entity name="r_buffer" class="util.ExecActor">
        <property name="comments" class="ptolemy.data.expr.StringParameter" value="Creates a raster map layer
showing buffer zones surrounding cells that contain non-NULL category values.">
            <property name="_style" class="ptolemy.actor.gui.style.NotEditableLineStyle"/>
        </property>
        <property name="commandline" class="ptolemy.data.expr.StringParameter" value="r_buffer [-qz] input=float
output=string distances=float[,float,...] [units=string]">
            <property name="_style" class="ptolemy.actor.gui.style.NotEditableLineStyle"/>
        </property>
        <property name="-q(Run quietly)" class="ptolemy.data.expr.Parameter" value="false">
            <property name="style" class="ptolemy.actor.gui.style.CheckBoxStyle"/>
        </property>
        <property name="-z(Ignore zero (0) data cells instead of NULL cells)" class="ptolemy.data.expr.Parameter"
value="false">
            <property name="style" class="ptolemy.actor.gui.style.CheckBoxStyle"/>
        </property>
        <port name="input_trigger" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="type" class="ptolemy.actor.TypeAttribute" value="boolean"/>
        </port>
        <port name="input_fn" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="type" class="ptolemy.actor.TypeAttribute" value="string"/>
        </port>
        <port name="output_fn" class="ptolemy.actor.TypedIOPort">
            <property name="input"/>
            <property name="type" class="ptolemy.actor.TypeAttribute" value="string"/>
        </port>
        <property name="distances" class="ptolemy.actor.parameters.PortParameter" value="{1.0}"/>
      <property name="units" class="ptolemy.actor.parameters.PortParameter" value="meters">
        <property name="_stringMode"/>
            <property name="style" class="ptolemy.actor.gui.style.ChoiceStyle">
                <property name="C0" class="ptolemy.kernel.util.StringAttribute" value="meters"/>
                <property name="C1" class="ptolemy.kernel.util.StringAttribute" value="kilometers"/>
                <property name="C2" class="ptolemy.kernel.util.StringAttribute" value="feet"/>
                <property name="C3" class="ptolemy.kernel.util.StringAttribute" value="miles"/>
                <property name="C4" class="ptolemy.kernel.util.StringAttribute" value="nautmiles"/>
            </property>
      </property>
        <port name="output_trigger" class="ptolemy.actor.TypedIOPort">
            <property name="output"/>
            <property name="type" class="ptolemy.actor.TypeAttribute" value="boolean"/>
        </port>
    </entity>
```

Fig. 3 MoML representation of executable actor corresponding to r.buffer

To correctly capture control options and parameters of a GRASS GIS module that an executable actor represents, “util.ExecActor” (c.f. Fig. 3) does much more work than “util.ConceptActor” (c.f. Fig. 2) which is mostly a placeholder. When an instance of ExecActor is executed (or “fired” in Ptolemy terminology), the name of the actor is obtained as the initial value of the command line string. Second, the actor checks all its parameters. If the names of the parameters begin with “-“, then they are treated as control options (c.f. rule 4). If the value of such a parameter is “true” then its name (such as “-a”) is appended to the command line string. Third, the actor

checks the input ports of the executable actor. If the width of the port is zero, which means no data is fed to the port (if the port is a ParameterPort rather than a regular port), the persistent value of the port is obtained. Otherwise the value is obtained from a regular port dynamically (c.f. rule 3). For the port values end with "_fn" or embraced by "{" and "}" pair, they are trimmed from the port values (c.f. rules 2, 7 and 8). The port names and the trimmed port values form "key=value" pairs and will be appended to the command line string. Once building the command line string is finished, a true value is send to the actor's output indicator (output_trigger) port to tell the workflow execution scheduler to execute next actor (c.f. rule 9). The command line string can either be used to invoke the GRASS GIS module within the actor execution process or output as part of the script file to be executed in a local or remote computation grid.

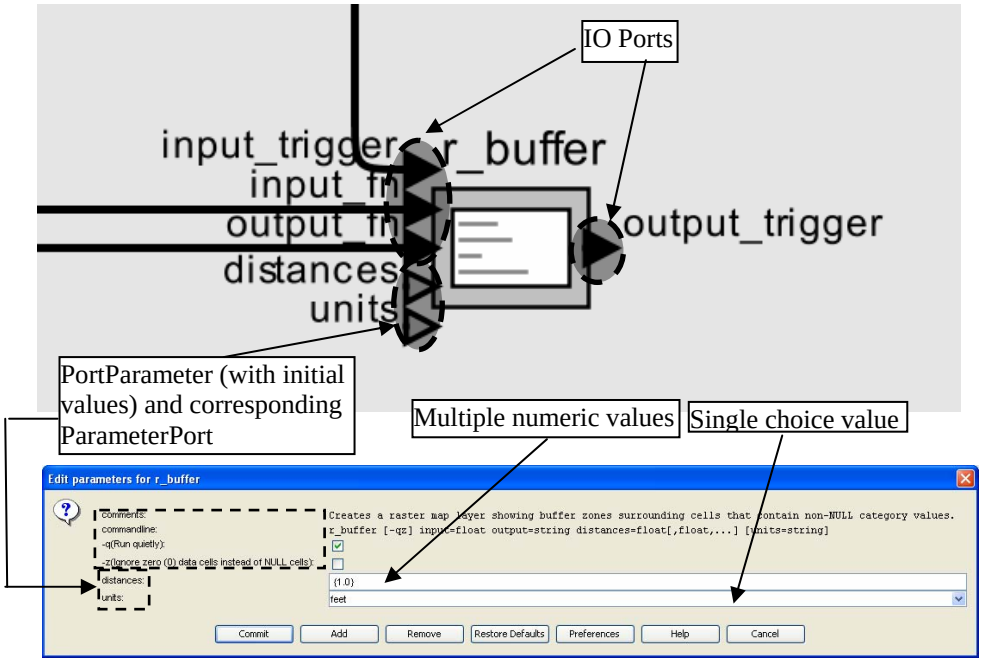

Fig. 4 Graphical representations executable actor r_buffer

4. Automatic Transformation

Supported by Kepler scientific workflow system infrastructure and the rules for mapping GRASS GIS command line modules to executable actors, we can now present the algorithm to automatically transform conceptual workflows to executable workflows.

1. For each conceptual actor in a conceptual workflow, find the corresponding executable actor by looking up its name in the executable actor library (c.f. rule 1).
2. For each connecting conceptual actor pair, find the port named "output_trigger" from the executable actor corresponding to the source conceptual actor and the port named "input_trigger" from the executable actor corresponding to the destination conceptual actor. Connect the two ports in the executable workflow (c.f. rule 9).
3. For each output port of the conceptual actors, find its connecting input port. If the names of the two connecting ports both ends with "_dt", find the ports with the same roots but ending with "_fn" in the corresponding executable actors, create a Const actor (representing a file name) and connect the output port of the Const actor to the input ports of the two executable actors in the executable workflow with a relation (c.f. rule 2).
4. For each input port of the conceptual actors, if the port name ends with "_dt" and there is a port in the corresponding executable actor whose name has the same root but ends with "_fn", create a Const actor (representing a file name) and connect the output port of the Const actor to the input port of the executable actor in the executable workflow with a relation (c.f. rule 2 again).
5. Layout the executable workflow.
    a. Executable actors are put at the same locations in the executable workflow as their corresponding conceptual actors in the conceptual workflow. The extent of the executable workflow on the composition canvas can be calculated.
    b. Put all the added Const actors at the left side of the executable workflow. All their x coordinates are assigned a fix number (e.g. 50). The y coordinate of the i^{th} Const actor can be computed as $i*h/n$ where h is the height of the previously calculated workflow extent at y direction and n is the number of added Const actors.
    c. For the added Const actors whose output ports connect more than one port, vertexes are added to the relation associated with the output ports. The added vertexes give more freedom to connect ports with straight lines and produce nicer workflow graphic representations (c.f. Fig. 5 in Section 5).

While the automatic transformation algorithm reduces most of the needed efforts of executable workflow composition (adding actors representing file names, change input/output data ports of conceptual actors to input file name ports of executable actors and connect them correctly), manual post-processing may be required for some complex conceptual workflows. We are in the process of developing more sophisticated algorithms to handle complex conceptual workflows.

5. Demonstration

The demonstrative example is a simple workflow that creates a convex hull from a point data set using GRASS GIS's *v.hull* module. The convex hull is then rasterized

using the *v.to.raster* module. Finally the *r.buffer* module is used to create buffers around the convex hull. The conceptual workflow (Fig. 5 top) is fairly simple and each of the three actors has one input port and one output port. The abstract nature of the conceptual workflow is idea for validation. We are working towards data types based semantic workflow validation based on some previous work ([5][6]).

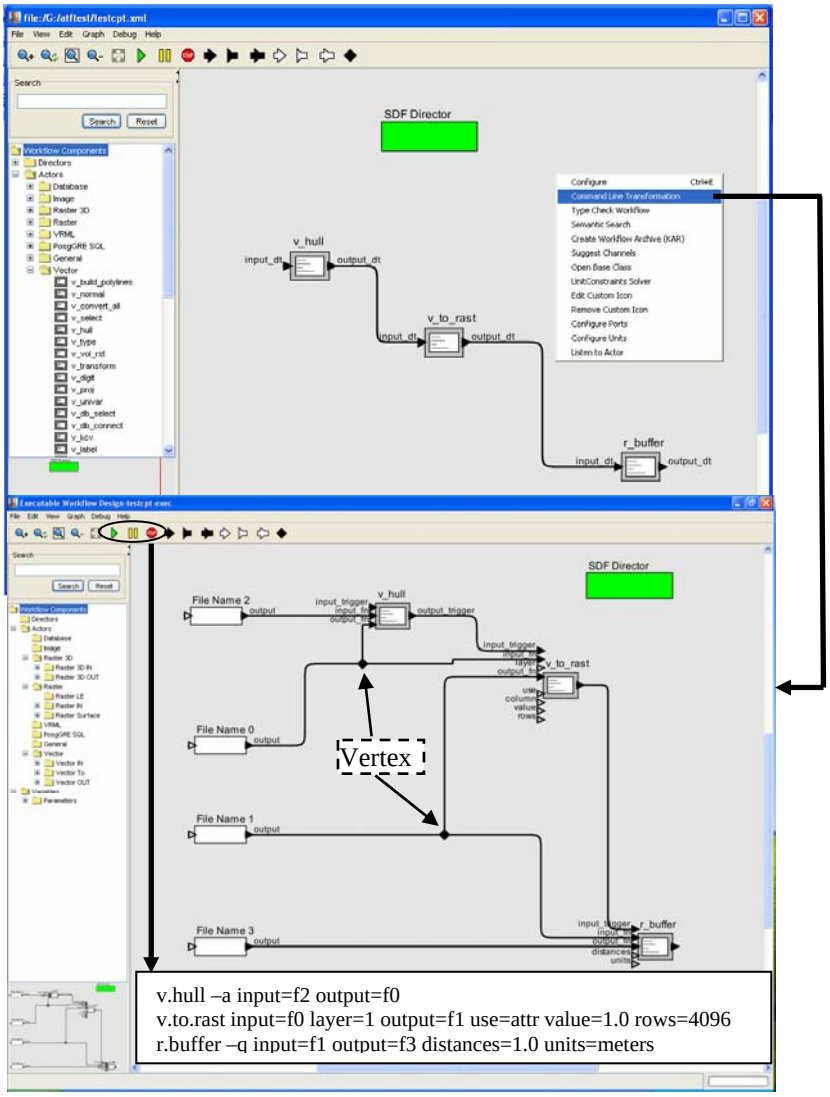

Fig. 5 Conceptual (top) and executable (bottom) workflows of the example

The derived executable workflow (Fig. 5 bottom) is more complex than the conceptual workflow. However, the parameters and the ports of the executable workflow are syntactically compatible with the parameters of GRASS GIS modules.

When the executable workflow is executed, GRASS GIS command line scripts can be generated from the executable workflow as shown in the lower part of Fig. 5.

6. Summary and Future Work Directions

We proposed a two-step approach to the composition of geospatial scientific workflows using open source GRASS GIS command line modules. Built on top of Kepler scientific workflow system infrastructure, we developed both conceptual and executable actor libraries for GRASS GIS command line modules. We also provided a practical algorithm to automatically transform conceptual workflows to executable workflows after users construct and validate the conceptual workflows. A demonstrative example was presented to show the feasibility of the proposed approach and the functionality of the preliminary implementations within Kepler.

For future work, we would like to develop more sophisticated automatic transformation algorithms to handle complex conceptual workflows. While Kepler supports semantic type checking for conceptual workflow design, integration is left for future work. Furthermore, although the implementations currently support GRASS GIS modules only, including modules from other GIS package (such as ESRI ArcGIS) is planned.

Reference

[1] GRASS Development Team, 2005. GRASS 6.0 Users Manual. ITC-irst, Trento, Italy. Electronic document: http://grass.itc.it/grass60/manuals/html_grass60/

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, S. Mock, 2004, Kepler: An Extensible System for Design and Execution of Scientific Workflows, the 16th International Scientific and Statistical Database Management Conference (SSDBM), 423-424

[3] B. Ludäscher, I. Altintas, A. Gupta, 2003, Compiling Abstract Scientific Workflows into Web Service Workflows, the 15th International Scientific and Statistical Database Management Conference (SSDBM), 251-254

[4] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (eds.), Heterogeneous Concurrent Modeling and Design in Java, http://ptolemy.eecs.berkeley.edu/ptolemyII/

[5] S. Bowers, B. Ludäscher, 2005, Actor-Oriented Design of Scientific Workflows, the 24th International Conference on Conceptual Modeling, 369-384

[6] J. Zhang, D. Pennington, W.K. Michener, 2005, Validating Compositions of Geospatial Processing Web Services in a Scientific Workflow Environment, the 3rd IEEE International Conference on Web Services (ICWS), 821-822