User Driven Workflows for the Collaboration of Grid Services

Günter Kickinger, Peter Brezany, and Ivan Janciak
Institute for Software Science
University of Vienna
Liechtensteinstrasse 22, A-1090 Vienna, Austria
Email: {brezany|janciak|gk}@par.univie.ac.at

Abstract – In the past five years 'The Grid' emerged as one of the most important new developments in building the computing and data management infrastructure for science, business, health, and society in the 21st century. This paper deals with the development and organization of the knowledge discovery processes in Grid environments, especially, with the collaboration and coordination of a set of OGSA based Grid services developed within our Grid knowledge discovery system called GridMiner. One of the kernel parts of GridMiner is an advanced workflow component, which can be also included into other Grid systems implementing applications having non-trivial workflow requirements.

I. Introduction

Grid computing has been identified as an important new technology by a remarkable breadth of scientific and engineering fields as by many commercial and industrial enterprises [1]. The Vienna GridMiner project [2] aims to extend the state-of-the-art Grid technology to a completely new societally important category of applications. It develops and thoroughly evaluates the novel concepts of knowledge discovery in large databases and other data repositories integrated into the Grid. The project focuses on data mining and On-Line Analytical Processing, two complementary technologies, which, if applied in conjuction, can provide a highly efficient and powerful data analysis and knowledge discovery solution on the Grid.

Grid services are designed to enable virtual and distributed organizations [7]. Every Grid service stands for a reusable resource within the Grid. GridMiner is a grid-aware service-oriented global infrastructure. If we analyze the anatomy of the knowledge discovery processes that are to be implemented by the GridMiner's services and their sequence organization, we realize that we need to extend the GridMiner by a special subsystem allowing us to combine the different services to a workflow of activities.

Such a combination of services is not needed for GridMiner only. Any service-oriented Grid can benefit from a language or an application which enables the description of collaboration of services too. We can identify two different approaches which will be necessary. A service-orchestration approach, where a new service is orchestrated by a combination of other existing services, and a user-defined collaboration approach for unique workflows, where a user can determine which services to involve.

The GridMiner architecture consists of a set of various Grid services, which are designed to support the single steps of the knowledge discovery process. The objective of this paper is to develop a control layer which is able to coordinate, manage and control the various services to support the extraction of new previously only implicit knowledge out of data.

In this article, we describe the structure and components of the knowledge discovery processes and their mapping onto appropriate Grid services. Further, we address the challenge of integrating these services and describing the various interaction between them, and how we can manage the resulting workflow. Our approach is based on the Open Grid Service Architecture (OGSA) [6]. GridMiner is the the first prototype of a Grid knowledge discovery system developed on top of OGSA.

The topics presented here are only a small and general excerpt of our work. A more detailed description can be found at www.gridminer.org.

II. The Knowledge Discovery Process

Knowledge discovery in databases (KDD) can be defined as *the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data* [4]. Often the terms *data mining* and KDD are used as synonyms. But principally, as described below, data mining is only one step within the whole KDD process (see Fig. 1), which concerns with applying appropriate algorithms to extract knowledge from a prepared dataset.

- *Cleaning and Integration.* Data in operational databases are typically not "clean". This means that those databases contain errors, due to wrong inputs from users or application failures. Besides, this data may be incomplete, and essential information may not be available for some attributes. Hence, for data mining and other analysis tasks, it is necessary to "clean" data and to integrate the data into one common dataset.
- *Selection and Transformation.* The next step of KDD will be to choose an appropriate subset of the integrated data and perform some necessary transformations. Many data mining algorithms work only on specially transformed data. One example is a neural network algorithm which needs the input in a distinct interval like *[0,1]*.
- *Data Mining.* This step involves the application and parameterization of a concrete data mining algorithm, to search for structures and patterns within the dataset, like a *classification, clustering, association, characterization,* or *comparison* algorithm.
- *Evaluation and Presentation.* In the last step, the results of data mining are evaluated by distinct interestingness measures. Data mining algorithms deliver a set of patterns. Often these patterns are not interesting to a user. Hence, measures like support and confidence discriminate interesting from not interesting patterns. Of course the results of data mining have to be

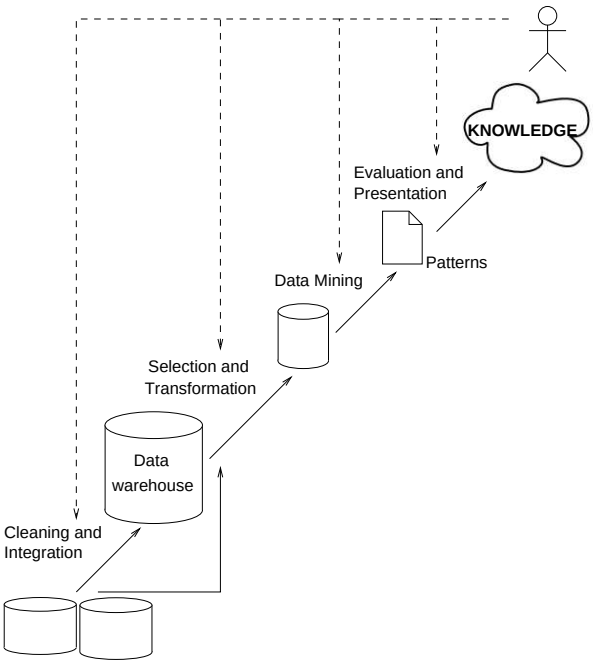

Fig. 1. The KDD Process [9]

prepared in some appropriate way to present them to a user. Various presentation forms exist, like tables, charts, or more sophisticated ones.

III. Architecture of the GridMiner System

GridMiner is a set of OGSA conformous services with well-defined interfaces, that are used as needed to assemble applications on top of them. These services cover the steps of the KDD process horizontally and are supported or used by services below and above them - see Fig. 2. The services of the GridMiner Core level support the KDD process directly, and hence they are used by the client.

- *GridMiner Data Integration Service (GMDIS).* Data integration refers to transferring data subsets from multiple data sources into usually one other data source that is either more efficiently accessible or more analysis dedicated. GMDIS is responsible for secure, reliable, efficient management and operation of the necessary data transfers within grid environments.
- *GridMiner Preprocessing Service (GMPPS).* We subsummarize under this term preprocessing activities (besides data integration) that are performed before the data mining step like data cleaning, normalization, selection, reduction etc. The Preprocessing Service is a service that can easily be extended by additional algorithms.
- *GridMiner DataMining Service (GMDMS).* This service is one of the major core components of the GridMiner system. It is an extensible framework providing facilities that all data mining algorithms usually need, and data mining application developers can easily plug their algorithms and tools in.
- *GridMiner OLAP Mining Service (GMOMS).* It provides algorithms to perform data mining on OLAP cubes. This is a service which works very closely together with an OLAP engine providing, for example, 'drill up' and 'drill down' operations, which allow to analyze datasets at different abstraction levels.
- *GridMiner Dynamic Service Composition Engine (GMDSCE).* This service allows the execution of complex, dynamic workflows for distinct knowledge discovery scenarios. The workflow itself is specified by an XML-based language designed within our project. GMDSCE is described in more details in Section V.

IV. Collaboration of the GridMiner Services

The previous section showed the base services provided by the GridMiner. Each of these services is able to perform one step within the knowledge discovery process. Now we investigate how the different services work together to support the whole knowledge discovery process and not only one of its parts.

The services themselves do not communicate with each other. No service is aware of other existing services. Hence each of the services is able to run completely independently. To support the individual steps of KDD process, the output of the first service serves as input for the second service. The independence of the various services also allows a parallel execution without any communication overhead. This results in an improvement of performance.

The data themselves of course, are not sent via the service parameters, since they can get very large. The input parameters and the service results contain only references to files or other resources (like references to data services). So data is not transmitted to the services. Every service can choose itself how it would handle the data access (either copying data by GridFTP or remote access by intelligent selection of the needed data rows). Data is not only physical data stored in files or databases. Input or output data sources can also be represented as access services. So some middleware service can provide the data.

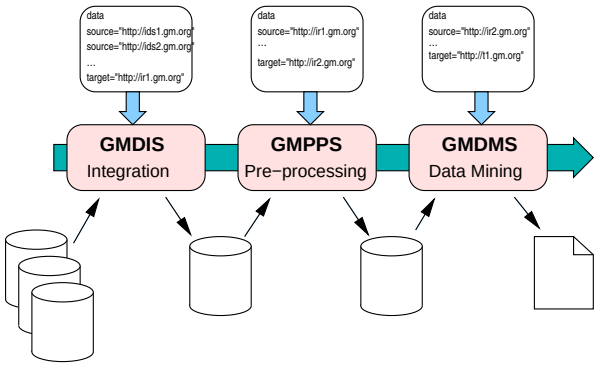

Fig. 3. Example of one KDD process with GridMiner

Fig. 3 shows an example how various GridMiner services can be connected to perform one KDD process. The services do not interact with each other. The only existing connection are the different data sources. Every service has a set of parameters as input. These parameters tell the service for example where to get the input data from, where to store the results and so on and what distinct task to do. The data sources depicted in this figure does not need to be data from a database. So it is possible, that a service uses a common file as input or output data, or even that this data can be an own service. However the service parameters contain information, about the location and type of data.

In this scenario, the client wants to do, for example, clustering on a specific data set. The data is distributed so it would be first necessary to integrate the different data

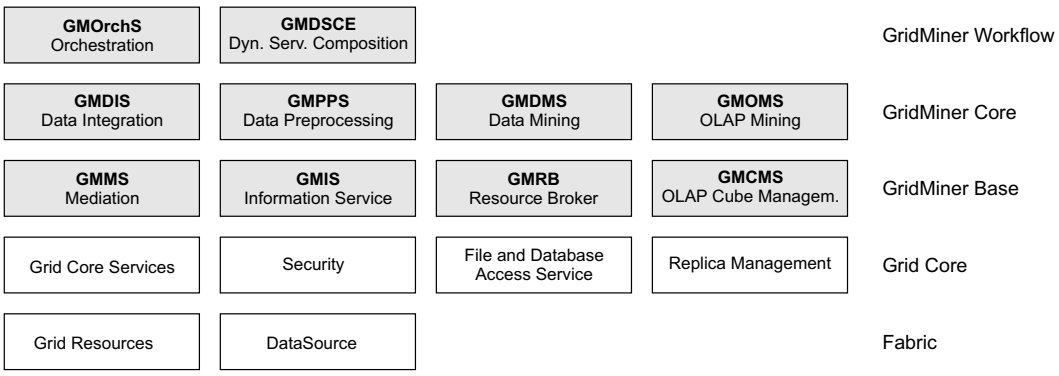

Fig. 2. GridMiner components

sources into one data set. The parameters tell GMDIS what to do, so they specify where to find the input data sets and where to write the results. The resulting data set is used as input for the GridMiner Pre-Processing Service, which does for example some data cleaning. Next, GMDMS is started, which tries to find clusters. Again the service is controlled via its parameters. So the parameters include the references to data, information about the algorithm and some interestingness measures for the discovered patterns.

The example above illustrates only one simple scenario. The sequence of the service calls can be far more complex. It is very likely that distinct services are executed in parallel. One example are sampling methods before integration of data. Those sampling methods draw random samples from huge data amounts. If a user wants to mine on more than one data source, it is better to apply sampling before data integration is performed. Or consider that incomplete raw data records shall be ignored. Obviously it is better to execute a preprocessing service for each data source before integration. So the data integration service may need to process less data. It is obviously, that those preprocessing services are executed in parallel and not sequentially. For data mining on top of OLAP we need to replace the GMDMS by an other service called GMOMS (GridMiner OLAP Mining Service) and to add a service which creates the cubes (GridMiner Cube Management Service). Sometimes the user wants to apply different algorithms for the same methods, to look which one best fits to the problem solved and compare the results. Often it is useful to call more than one pre-processing service (E.g., the first one does cleaning, the second one data transformation, etc.). So we have many possible scenarios for constructing a workflow which supports knowledge discovery.

V. Workflows

In the previous sections we investigated how the underlying service can be deployed to cover the whole knowledge discovery process, both on top of flat tables and on top of OLAP. The execution of this knowledge discovery process can take from a few minutes to several hours or even longer. If a client application, located for example on a user's workstation, controls the job and the job would be terminated, following tasks would never be executed. The results of the already executed services would have been lost and the job would be aborted. KDD can include several concurrent activities, e.g. several classification models can be created in parallel and then sequentially evaluated. If a client implements all this functionality, it will become very complex and very "thick". Thus we need a service which is able to handle those long running, complex jobs and executes them on behalf of an execution plan.

Our goal is to develop a *workflow engine* which *instantiates*, *executes* and *coordinates* a set of *OGSA Grid services* according to a predefined *execution plan*.

Two different approaches for implementing a workflow engine can be identified (Figure 4). These approaches and their differences are discussed in more detail in the following sections.

A. Static Service Composition by Service Publisher

Figure 4 (a) gives a schematic overview of *static service composition*. Service A and Service B are two already existing services. Now a service publisher recognizes that it would be useful to combine these two services and to provide an orchestrated service (Service [AB]). He does so by writing a workflow file, which describes the interaction of the two services involved. Additionally he creates a new WSDL file, which describes the ports and operations of the orchestrated service. Now he installs this service within a workflow engine. The result of this process is a new service, which can be used by a client like a common web service. Workflow languages for Web Services like BPEL4WS [13] are using this strategy.

B. Dynamic Service Composition by Service Consumer

GridMiner requires dynamic workflows, which are not reusable in the sense that a workflow is pre-described by a workflow description language once and installed to a workflow engine. Since every knowledge discovery process requires a unique specific workflow, the workflow engine does not "construct" an own orchestrated service.

There is also another reason why the composition to a new service is not useful. The interface for accessing the workflow service must be static. If a service publisher orchestrates a new service, the interfaces and operations of the new orchestrated service can be different from those of an old one. A client written to use the old service possible cannot operate on a new composed service or the implementation of a client would result in enormous complexity, because it cannot make use of proxy classes since the service interfaces would change permanently. So, whereas the workflows are dynamic the interface to access the service instance has to be static.

Figure 4 (b) gives a schematic overview of *dynamic service composition by a service consumer*. Services A, B,

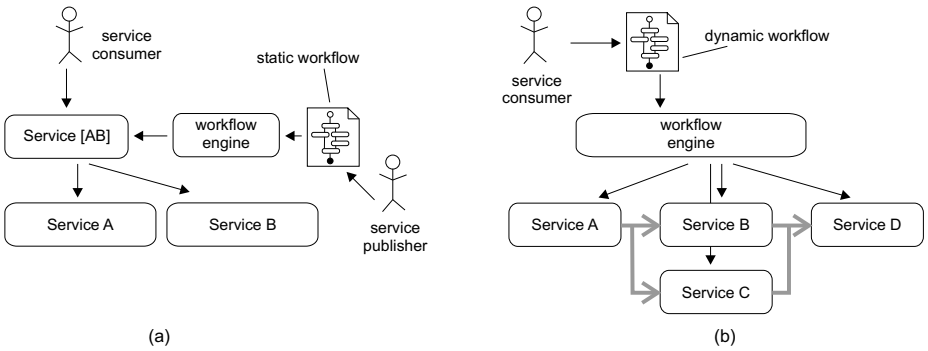

Fig. 4. A static workflow specified by the service provider that is executed each time in the same way is shown in (a). Contrarily a service consumer specified workflow with a more complex scenario is shown in (b).

C, and D are already existing services. Now the service consumer specifies in a workflow file in which order he wants to execute the services, i.e. the number of the instances to create from each service, which services can be executed in parallel. In the example depicted in Figure 4 (b), the user specifies that service A has to be executed first, then service B and C have to be executed concurrently, and as the last step service D has to be called. He also specifies the concrete values of the arguments for each of the involved services. Than the service consumer sends this workflow description to a workflow engine, which executes the workflow.

C. Dynamic Service Control Language

The developed solution is based on the "Dynamic Service Composition by a Service Consumer" approach identified in the previous section. The user decides which services are needed to fulfill the knowledge discovery process and describes the process in a workflow description file. This description is done by the *Dynamic Service Control Language* (DSCL), an XML notation, which has been specially designed to support dynamic and complex workflows. DSCL is a low level language which utilizes the idea of Grid services. It allows the creation of Grid service instances by their factories, it can invoke operations on Grid service data elements, it can process notifications, and it supports sequential and concurrent execution.

Unlike common languages like BPEL4WS, which aim to provide a new orchestrated service composed out of other services—DSCL is execution centric, that means a DSCL document is sent to a workflow engine and this engine executes all operations of the underlying Grid services with the parameter values specified.

DSCL can always be used when a client wants to start a workflow of one or more Grid services. So the workflow composition is done by a client and is not pre-defined. This is necessary if the client does not know which services it will call and in which order they have to be executed in advance, that means at compilation time. In most cases a client will use stubs and proxies to communicate with a Grid service. So it has already to "know" at compilation time which services it will use later.

Using DSCL the client can call any Grid service it "likes" or which a user tells it. The only things which need to be specified are the location of the service, the desired operation and parameters of the operation. Therefore the client is very flexible too; Grid services can easily be exchanged without changing the implementation of the client. The use of DSCL is very similar to a common programming language; first of all the variables have to be declared and initialized with values. These variables serve as input parameters to a service call, or as a storage for the returned value of a service invocation. Second the control flow has to be specified. Within the control flow it is possible to:

- create new Grid service instances
- invoke synchronous and asynchronous operation on Grid services
- query service data elements from a Grid service
- destroy Grid service instances
- execute a set of activities concurrently
- execute a set of activities sequentially

A workflow described by DSCL consists of two important sections: the <**variables**> section and the <**composition**> section. <**variables**> pre-defines all parameters later used by the involved services. These include both, input parameters and output parameters A concrete parameter is defined in scope of the <**variable**> tag. If the parameter content is known at built time of the workflow, it is necessary to initialize it, that means to fill in concrete parameter values. The initialization of a parameter follows the SOAP [16] RPC style. Sometimes, not all of the parameter values are known in advance. This happens, if the values are created by a service which will be called earlier in the workflow. In this case it is only necessary to declare the variable without any initialization.

The <**composition**> section describes the workflow to be executed. We define our workflow as a set of *activities*. We distinguish two kinds of the activities. First of all, we have *control activities*. These are activities like *"start parallel execution"* (<**parallel**>) or *"start sequential execution"* (<**sequence**>). These control activities consist of other activities. DSCL defines activities for creating a new service by a factory or destroying it (<**createService**>,<**destroyService**>), calling an operation (<**invoke**> and querying the service data elements of a grid service instance (<**querySDE**>). All of these activities make use of parameters described within the <**variables**> section. The example below shows the description of a very simple workflow by DSCL.

```
<dscl:dscl xmlns:dscl="..." xmlns:xsi ...>
    <dscl:variables>
        <dscl:variable
            dscl:name="param1">
```

```
            <ns1:value
                  xsi:type="xsd:int" ...>4711 </ns1:value>
        </dscl:variable>
    </dscl:variables>
    <dscl:composition>
        <dscl:sequence>
            <dscl:createService
                  dscl:factory-name="..."
                  dscl:instance-name="test"/>
            <dscl:invoke  dscl:instance-name="test"
                  dscl:portType=""
                  dscl:operation="setIntValue"
                  dscl:inputParam="dscl:param1"
                  dscl:outputParam="dscl:..."/>
        </dscl:sequence>
    </dscl:composition>
</dscl:dscl>
```

D. Dynamic Service Composition Engine

The *Dynamic Service Control Engine* (DSCE) is an implementation of a workflow engine, which processes documents based on the *Dynamic Service Control Language*. The DSCL document is parsed by DSCE and the workflow is executed, that means that each of the activities is processed according to the DSCL specification. DSCE is based on the Open Grid Services Architecture (OGSA) [6] and implemented with the *Globus Toolkit 3.0* (GT3) [8]

Since in most publications, a workflow engine is a service, which allows the composition of services to a new orchestrated service [13], [12], [11], [14], we use the term Dynamic Service Control Engine (DSCE) to make our dynamic, service consumer related approach more expressive. DSCE is developed for GridMiner, but does not necessarily depend on it. It is implemented as an application-independent service, which is able to execute a dynamic process, consisting of a set of OGSA Grid services. So every system consisting of different Grid services, which are executed in different orders, and/or making use of parallel processing, can benefit from DSCE.

DSCE is implemented as a stateful, transient OGSA Grid service. That means DSCE first of all provides service data elements which describe the state, and the results of all involved activities. In data mining the user is not only interested in final results. If the results are not meeting his expectations, the user has to take a look at the intermediate results. Hence he or she must be able to analyze the whole workflow. Second, DSCE is transient. To use the functionality of DSCE a new service instance has to be created first. When to use an existing service instance or when to create a new one depends, but it is recommended to create a new service instance, if a completely new and different workflow is issued. If the workflow is based on an old execution, it is better to change the workflow description and restart the engine. The workflow itself is described by DSCL.

To support both batch and interactive processing, DSCE provides:

- *Independent processing of a workflow described in DSCL.* This feature is necessary to support batch processing. A client application can ask the engine to start the workflow, so the execution of the workflow begins. Now the execution does not depend on the client. This is very important, since the client does not need to run any longer. So the client can terminate and the execution is not influenced at all. The client can connect later and investigate the state of the workflow. Even if DSCE has finished the workflow execution, the service itself will be alive. It is possible for the client to fetch all results and all information of any activity that has been executed.
- *The provision of all intermediate results from the services involved.* The states and results of all activities can be queried by a client at any time. Therefore it is also possible to query them even if the execution of the workflow is going on. Moreover the client gets informed whenever the state of an activity or the state of the workflow has changed.
  So whenever an activity has been started, has been finished, or has failed the client is notified.
- *The possibility to change workflow at run time.* Suppose the results of a "run" does not satisfy the user, so he analyses the intermediate results. If he recognizes, for example, that he applied a wrong algorithm or wrong parameters, he can change them in the workflow description document and start the workflow again.
- *The possibility for a user to stop and resume a workflow.* If the user recognizes—even during execution—that the results are not confident (The client can query the results any time like explained above), there is no need to execute the rest of the workflow, so the user can stop the execution.
  A user is also allowed to resume a stopped or erroneous workflow. The execution will be started, where it has been stopped before or where the error occurred.

Because DSCE is implemented as a stateful transient OGSA Grid service, first of all a service factory is needed which is responsible for creating one or more DSCE services. The factory is - by its definition - a persistent service; this means that it is always alive as long as its container is running. A client can use the factory to create a concrete instance of the DSCE service. This factory-instance concept allows the client to specify the lifetime of the instance. It can specify lifetime parameters at creation time and therefore transfer the responsibilities to the container, or which would make more sense in case of DSCE, it is responsible for destroying the service itself. Once a service instance is created, it is running within the container and can be accessed by any client via its address (the Grid service handle). Hence it is possible, that a client creates the DSCE service instance, gives orders to it and then quits without destroying the DSCE service instance. The instance stays alive and continues its processing, even if there is no client connected to it. The client can reconnect to the service and look at the progress done so far.

DSCE also implements the *Pull Notifications* model [15]. A client can subscribe to a notification sink and is notified by the DSCE service in the following situations:

- The status of an activity has changed. For example an activity has finished, an error occurred and so on.
- The status of the workflow has changed. For example if the processing of the whole workflow has been finished.

VI. Related Work

The first proposal of an architecture for performing data mining on the Grid was published in [3]. M. Cannataro

and D. Talia, present design of a Knowledge Grid architecture based on the non-OGSA-based version of the Globus Toolkit. BPEL4WS 1.1 (Business Process Execution Language for Web Services) [13] is the actual standard, which can describe compositions of Web Services. The Grid Services Flow Language [11] intends to do the same for Grid Services. GSFL is based on the so called Web Services Flow Language [12], a predecessor of BPEL4WS, published by IBM. Those flow language specifications have all the same target: they are describing a business process built up of various web services. This description then serves as input for a workflow engine like BPWS4J [10] (an engine for BPEL4WS developed by IBM).

VII. Conclusions and Future Work

Within this paper a new infrastructure to describe and control the collaboration of a set of OGSA conformous Grid services has been evolved. We started with the investigation of the knowledge discovery process and discovered that within the Grid no acceptable technology is available to support user-driven processes.

Dynamic Service control (DSC) is an approach which contributes to improve this situation. It specifies an XML-based language—the *Dynamic Service Control Language* (DSCL)—which allows a user or—more precise—a service consumer to describe a workflow as a collaboration of various OGSA Grid services. DSCL is not only designed to specify the knowledge discovery process, but also to specify user-driven workflows in general. In addition to DSCL we specified, designed, and implemented a prototype of a workflow enactment engine called *Dynamic Service Control Engine* (DSCE). DSCE executes a workflow described by DSCL according to the specification. The prototype is an interactive, asynchronous OGSA Grid service allowing a user to send commands anytime, and hence is much more than a batch processing system.

DSCE can help to reduce the complexity of Grid client applications significantly. Clients do not have to deal with topics like concurrent execution, and synchronization of asynchronous invocations. Additionally a client is rather independent from the Grid services available. So it is considerable, that the user (of the client) can decide which Grid services to use. The client implementation does not have to be changed at all.

The emphasis of DSC was to support workflows which can describe the knowledge discovery process and to serve as a control layer in the GridMiner system. To become a general approach the usability within other applications has to be investigated. It may be the case that additional requirements are identified and DSC has to be extended, for example, by error handling, loops, and branches. In all cases DSC provides a fundamental approach, which can be the base for further development. Additionally it will be necessary to extend DSCE by technologies which are able to optimize the execution of workflows. One example may be the inclusion of a simple caching algorithm, so that DSCE will be able to "remember" to already executed workflows and intermediate, reusable results. It would be also interesting to design a distributed, parallel architecture of DSCE and investigate its performance. DSCE was tested and implemented using SOAP over HTTP. If another, more performant protocol shall be used, it has to be investigated whether and which changes have to be made. In January 2004 a new infrastructure bringing Web services and Grid services together—the Web Services Resource Framework [5]—was proposed. The work presented here is closely related to OGSA/OGSI. So it has to be investigated which changes have to be made to adopt DSCL to the WS-Resource Framework.

As investigated in this paper there exists another approach for solving the workflow problem—the orchestration of Grid or Web services. Within this work it was shown that this approach is not enough to solve all problems dealing with workflows and processes. DSCL can be an initiation to consider these aspects and maybe create a universal workflow language.

References

[1] F. Berman, G. Fox, and A. J. G. Hey, *Grid Computing - Making the Global Infrastructure a Reality*. John Wiley, 2003.

[2] P. Brezany, J. Hofer, A. M. Tjoa, and A. Wöhrer, "GridMiner: An Infrastructure for Data Mining on Computational Grids," 2003.

[3] M. Cannataro and D. Talia, "Knowledge grid: An architecture for distributed knowledge discovery," *Communications of the ACM*, Jan. 2003.

[4] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From Data Mining to Knowledge Discovery: An Overview," in *Advances in Knowledge Discovery and Data Mining*, U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, Eds. AAAI Press/ The MIT Press, 1996, pp. 1–43.

[5] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, T. Storey, W. Vambenepe, and S. Weerawarana, "Modeling stateful resources with web services. version 1.0," http://www-fp.globus.org/wsrf/ModelingState.pdf, Jan. 2004.

[6] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The physiology of the grid: An open grid services architecture for distributed systems integration," January 2002. [Online]. Available: http://www.globus.org/research/papers.html\#OGSA

[7] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the Grid: Enabling scalable virtual organizations," Intl. J. Supercomputer Applications, 15(3), 2001.

[8] The Globus Project, "Globus Toolkit 3.0," http://www.globus.org.

[9] J. Han and M. Kamber, *Data Mining. Concepts and Techniques*. Morgan Kaufmann, 2001.

[10] IBM, "The IBM Business Process Execution Language for Web Services Java Run Time (BPWS4J)," http://alphaworks.ibm.com/tech/bpws4j.

[11] S. Krishnan, P. Wagstrom, and G. Laszewski, "GSFL: A Workfbw Framework for Grid Services," http://www.globus.org/cog/papers/gsfl-paper.pdf, 2002.

[12] F. Leymann, "Web Services Flow Language (WSFL 1.0)," http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, IBM Software Group, May 2001.

[13] BEA Systems, IBM, Microsoft, SAP, and Siebel Systems, "Business Process Execution Language for Web Services," http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, May 2003.

[14] S. Thatte, "XLANG: Web Services for Business Process Design," http://www.gotdotnet.com/team/xmlwsspecs/xlang-c/default.htm, 2001.

[15] S. Tuecke, K. Czaijkowski, I. Foster, *et al.*, "Open Grid Services Infrastructure (OGSI) Version 1.0 (draft)," Apr. 2003.

[16] W3C, "Simple Object Access Protocol (SOAP) 1.1," http://www.w3c.org/TR/SOAP, 2000.