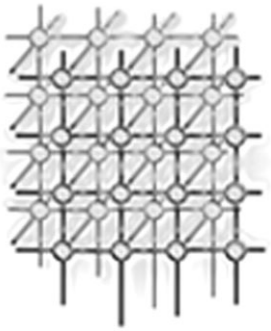
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; **18**:1317–1331
Published online 13 December 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.995

Toward a search architecture for software components



Fabrizio Silvestri[1,*,†], Diego Puppin[1], Domenico Laforenza[1] and Salvatore Orlando[2]

[1]*HPC-Lab, ISTI-CNR, Via G. Moruzzi 1, 56124 Pisa, Italy*
[2]*CS Department, University of Venice, Via Torino 155, 30172 Mestre, Italy*

SUMMARY

The Grid and its related technologies enable large-scale sharing of resources of various types. We envision that in the near future applications will be completely built in a bottom-up fashion using software components deployed on various locations and interconnected to form a workflow graph. In this paper, we make some proposals on the design of a component search service, enabling users to locate the components they need to deploy an application. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: Grid computing; workflows; component rank; component search; PageRank

1. INTRODUCTION

More and more, today's business and scientific applications are built as a complex network of services offered by different providers, on heterogeneous resources, constrained by administrative problems when crossing the borders of different organizations. Applications are no longer monolythic kernels running on a large computing cluster, but rather dynamic collections of computing entities.

This makes the age-old problem of component searching more important and central than ever: developers must be able to find the correct software component for their needs, choosing from a variety of vendors, service providers, computing departments, outsourcing firms. While FORTRAN programmers had to choose from a palette of library calls for their tasks, now Grid developers can potentially make use of any computer on the Internet.

Component searching for Grid-aware applications plays two major roles.

1. At design time, programmers want to find the component that best fits with the structure of their application. The component must be the most appropriate and trusted for the job at hand.

*Correspondence to: Fabrizio Silvestri, HPC-Lab, ISTI-CNR, Via G. Moruzzi 1, 56124 Pisa, Italy.
†E-mail: fabrizio.silvestri@isti.cnr.it

Received 1 June 2004
Revised 15 March 2005
Accepted 27 April 2005

2. At run time, the framework may want to substitute a component that performs too slowly or incorrectly with an equivalent component from a different vendor or provider. This is fundamental for the application to survive in a very dynamic environment like a computing Grid.

The increasing popularity of component-based frameworks for Grid programming (e.g. [1]) is accelerating the growth of the number of components available to developers. So, we envision the existence of a marketplace where developers can gather the components for their applications. In this market, the same kind of service can be sold by different vendors at different prices and with varying quality.

It is clear that one of the most challenging goals to pursue will be to find the most suitable components for each user's needs. As far as we know, there has been limited effort in the Grid research community towards this goal. In this paper, we are going to discuss the challenges we have to face in designing a search service for locating software components on the Grid.

The specifications of our search engine rely heavily on the concept of the *ecosystem of components*. The *ecosystem* is the set of connections among different services, competing for visibility and user's adoption, and cooperating within larger applications. When a complex component uses smaller components, this creates a sort of link between the two of them. Our vision is close to that of Eco-Grid, as described in [2].

This concept can be easily compared to the well known concept of the Web. Under this vision, a software component can be compared to a Web page, and an application built by composing different blocks can be seen as a Web site (i.e. a composition of different Web pages). From the application perspective, each part can be either a component available locally (i.e. a *local* Web page), or a remote component (i.e. a *remote* Web page). In addition, the links interconnecting Web pages can be compared to the links indicating interactions among components of the same application.

The main potential of this model is that developers can make publicly available the relationships between the different components involved in their applications. We believe that there are many reasons why they would do so.

- First of all, we believe that Open Source Grid applications will naturally appear. Their code could be made available through portals such as SourceForge[‡], GridForge[§], or FreshMeat[¶].
- Moreover, on the Web there are many examples of popular services that publicize their use of other important and effective services, therefore the same will happen for Grid applications, in order to attract users' trust: AOL, for instance, claims that it uses the Open Directory Project (ODP [3]) as its backbone for offering its search service. Translated to the Grid, the importance of an application could grow when it uses another popular component.
- Lastly, by publishing the links to external components, an application developer can support or promote standards and competition among vendors. This could have a very high commercial value.

[‡]http://www.sourceforge.net.
[§]http://forge.gridforum.org.
[¶]http://www.freshmeat.net.

In other words, *we want to use the composition graph of Grid applications to rank software components.* The idea is rather simple: the more a component is referred to by other applications, the more important it is considered. This concept is very close to the well-known PageRank [4] measure used by Google to rank the pages it stores.

In this introduction, we did not specify precisely the meaning of service or software component. In general terms, 'a component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A component can be deployed independently and is subject to composition by third parties' [5]. The reader familiar with Globus and WSRF can think of a software component as a service available on the Grid in the form of a Grid service. This is clearly only one example. In our experiments, we used Java classes due to their simplicity and high-quality standard of documentation. Other authors used ActiveX applets, or refer to the CORBA Component Model and the Common Component Architecture.

The rest of this paper is structured as follows. The next section discusses the concept of workflows and compositions. Then, Section 3 describes a number of studies focused on search engines for software components. Section 4 presents our ideas on search engines for components and introduces our prototype work. Finally, in Section 5, we conclude.

2. WORKFLOWS

Modern Grid application programming relies heavily on the concept of workflow. In general, a workflow can be described as 'a process description of how tasks are done, by whom, in what order and how quickly. Workflow can be used in the context of electronic systems or people, i.e. an electronic workflow system can help automate a physician's personal workflow, or to perform the so called in-silico experiments'[‖].

In the Grid research community, workflows are seen as a viable technique to specify applications. Coordination languages are the languages used to specify workflows. They express the interactions among processes.

Research in workflow covers many interesting aspects. For instance, there are many researchers that are looking for novel models to effectively represent workflows. Others focus on the way programmers can specify applications (i.e. the language that can be used), or perform studies on how workflows can be effectively annotated (possibly by using semantic ontologies) in order to make their management easier.

In BPEL4WS [6], workflows are described using a language for the formal specification of business processes and business interaction protocols. The main disadvantage of BPEL4WS is that it is a very low level language and it requires too much user effort in order to specify even a very simple workflow. Many projects try to solve this issue, and mainly two different ways are followed: the first is to try to simplify BPEL4WS by creating a new higher-level language (possibly inheriting some features from BPEL4WS); the second is to build a Graphical User Interface (GUI) that helps users specify workflows in a visual manner and generates appropriate BPEL4WS code.

[‖] From http://pip.med.umich.edu/glossary/index22.htm.

Obviously a combination of these is possible. For instance, Taverna [7], developed for the *my*Grid project, provides both a language and the software tools to facilitate the use of workflow and distributed computing technology within the e-Science community. Taverna consists of a very user-friendly visual toolkit (called *SCUFL workbench*) to create and enact workflows. The workbench is a sort of container which provides various views of the workflow being created and a controller to manipulate it. In addition to using BPEL4WS as the coordination language, they created *SCUFL* (Simple Conceptual Unified Flow Language), a user-oriented abstraction over general graph languages, which hides the details of service invocation and control flow.

A recent workshop, held in conjunction with the *Tenth Global Grid Forum (GGF-10)* [8], focused on the issues that need to be faced when dealing with Grid and workflows. The main concern of workflow in the Grid is that its specification should allow specific activities implemented by individual services to be exported and successively reused. It should also allow the exported activity to trigger a chain of other activities. When this is realized, there will also be a strong need for a service that helps users choose from the vast number of services that will be available.

3. RELATED WORK

In the last few years, thanks to technologies like the Internet and the Web, a number of interesting works [9,10] have studied the problem of discovering, indexing and searching for software components. Also, a number of interesting papers have analyzed new and existing solutions [11,12].

3.1. Spidering

The *Agora* components search engine is described in [10]. Agora is a prototype developed by the Software Engineering Institute at the Carnegie Mellon University. The object of this work is to create an automatically generated, indexed, worldwide database of software products classified by component type (e.g. JavaBean, CORBA or ActiveX control). Agora combines introspection with Web search engines in order to reduce the cost of bringing software components to, and finding components in, the marketplace. It supports two basic functions: the location and indexing of components, and the search and retrieval of a component.

For the first task, Agora uses a number of agents which are responsible for retrieving information through introspection. At the time the paper was written, Agora supported only two kinds of components: JavaBeans, and CORBA components. One of the most interesting features of Agora is its capability of automatically discovering the sites containing software components. The technique adopted to automatically find components is quite straightforward but appears to be effective. Agora simply crawls the Web, as a typical Web crawler does, and whenever it encounters a pages containing an `<APPLET>` tag, it downloads and indexes the related component.

Searching through this database is then performed using a keyword-based search, followed by a refinement process driven by the user.

3.2. Search

Several different approaches have been used to perform searches through a base of known components.

The *Odyssey* search engine [9] is an agent system responsible for domain information (i.e. domain items) search within the Odyssey infrastructure. It is composed of an interface agent (IA), filtering agents (FAs), and retrieval agents (RAs). The IA is the agent responsible for displaying the search results according to the users' profile. These profiles are modeled by identifying groups of users with similar preferences, stereotypes and so forth. The FAs match user keywords and the textual description of each component, returning those with the greatest number of occurrences of user keywords. Finally, the RAs are the agents responsible for searching for relevant components among different domain descriptions.

Prospector** is a search engine able to seek out code examples that use any or all of J2SE 1.4, Eclipse 3.0, and Eclipse GEF (Graphical Editing Framework) code. Prospector searches the graph for paths from the 'have' class to the 'want' class and then converts the paths into legal Java source code. This is a very interesting strategy to choose components that are the best match in a complex graph of component connections.

The Knowledge Grid project [13] at the University of Calabria also has an interesting strategy. It offers a development environment, called VEGA, where the user can sketch a Data Mining application to be run on a set of resources known as the Knowledge Grid. Using an ad hoc ontology for data-mining application (representing data sources, algorithms and so on), the system can show users a set of databases and tools that they can utilize to build their application. Our approach differs in that we want to limit the introduction of standards (i.e. ontologies) from above, but rather we want to utilize naturally emerging social patterns and links among existing software.

Within the myGrid project *Feta*, a Grid component discovery service has been developed. The main context from which Feta arises is bioinformatics [14]. Applications related to this field are characterized by the use of computational and mathematical techniques to store, manage and analyze the data from molecular biology in order to answer questions about biological phenomena. The main concern of Feta is to answer queries such as 'who has performed an experiment x, when, where and why?' or 'where can I find a service suitable to compute experiment x?', etc. The main innovation of Feta is the use of semantic information to annotate and describe a service. The myGrid project is focused on bioinformatics and so is the taxonomy developed using the DAML-S formalism. In fact, it starts from the general assumption that many service providers would be willing to adopt a 'standard' representation for data and services. The main disadvantage, though, is that such an approach cannot be used to describe components in general because of the need to have a taxonomy available for doing so. That is the reason why we have decided not to use semantic information within our discovery engine. However, the two approaches are not exclusive and can be integrated whenever necessary.

3.3. Ranking

In [12], the authors cite an interesting technique for ranking components within a given set of programs. Ranking a collection of components simply consists of finding an absolute ordering according to the relative importance of components. The method followed by the authors is very similar to the method used by the Google search engine to rank Web pages: PageRank [15]. In ComponentRank, in fact,

**http://snobol.cs.berkeley.edu/prospector-bin/search.py.

the importance of a component[††] is measured on the basis of the number of references (imports, and method calls) made by other classes to it within the given source code.

3.4. Comparative works

In [11], Frakes and Pole analyze the results of an empirical experiment with a real component search application, called *Proteus*. The study compares four different methods to represent reusable software components: attribute value, enumerated, faceted, and keyword. The authors tested both the effectiveness and the efficiency of the search process. The tests were conducted on a group of 35 subjects that rated the different methods used, in terms of preference and helpfulness in understanding components. Searching effectiveness was measured with recall, precision, and overlap values drawn from the Information Retrieval theory [16]. The most important conclusion cited in the paper is that no method performed more than moderately well in terms of search effectiveness, as measured by recall and precision.

4. A SEARCH ENGINE FOR SOFTWARE COMPONENTS

To date, we can observe that there has been very limited work on Grid-specific programming languages. This is not too surprising since automatic generation of parallel applications from high-level languages usually works only within the context of well-defined execution models [17], and Grid applications have a more complex execution space than parallel programs. Some interesting results on Grid programming tools have been reached by scripting languages such as Python-G [18], and workflow languages such as DAGMAN [19]. These approaches have the additional benefit that they focus on coordination of components written in traditional programming languages, thus facilitating the transition of legacy applications to the Grid environment.

This workflow-centric vision of the Grid is the one we are going to investigate in this work. We envision a Grid programming environment where different components can be adapted and coordinated through workflows, also allowing hierarchical composition. According to this approach, we thus may compose *metacomponents*, in turn obtained as a workflow that uses other components. An example of a workflow graph is shown in Figure 1. Even if this graph is flat, it has been obtained through composing different metacomponents, in particular 'flight reservation' and 'hotel reservation' components. As you may note, we have not chosen a typical *scientific* Grid application, but rather a *business-oriented* one. This is because, at the present time, we are witnessing the convergence of these two worlds, and because we would like to show that such Grid programming technologies could also be used in this case.

The strength of the Grid should be the possibility of picking up components from different sources. The question is now, where are the components located? In the following we are going to present some preliminary ideas on this issue.

[††]Only Java classes are supported in this version.

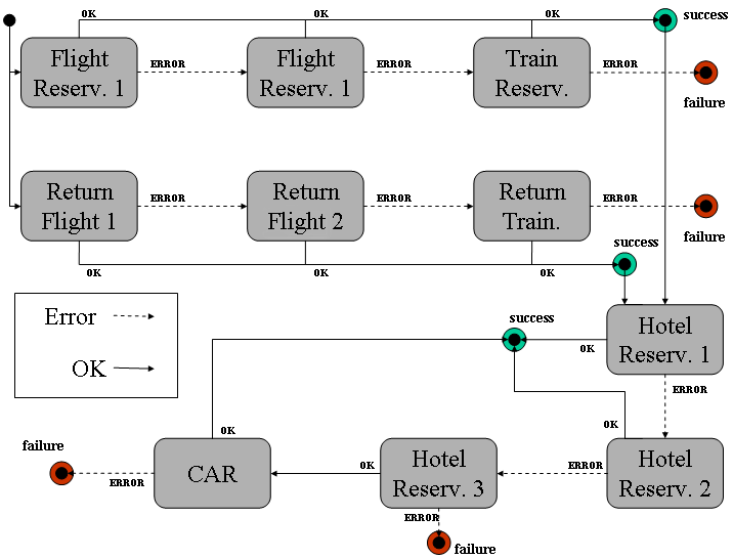


Figure 1. An example of a workflow-based application for arranging a conference trip. The user must reserve two flights (outward and return) before reserving the hotel for the conference. Note that, in the case that only the third hotel has available rooms, a car is needed and must be booked too.

4.1. Application development cycle

In our vision, the application development should be a three-staged process, which can be driven not only by an expert programmer, but also by an experienced end-user, possibly under the supervision of a problem solving environment (PSE). In particular, when a PSE is used, we would give the developer the capability of using components coming from:

- a *local repository*, containing components already used in past applications, as well as others we may have previously installed;
- a *search engine*, which is able to discovery the components that fit users' specifications.

Hence, the three stages which drive the application development process are:

1. application sketching;
2. components discovering;
3. application assembling/composition.

Starting from stage 1 (i.e. *sketching*), developers may specify an *abstract* workflow graph or *plan*. The abstract graph would contain what we call *place-holder* components and flow links indicating the way information passes through the application's parts.

A place-holder component represents a partially specified object that just contains a brief description of the operations to be carried out and a possibly inaccurate description of its functions.

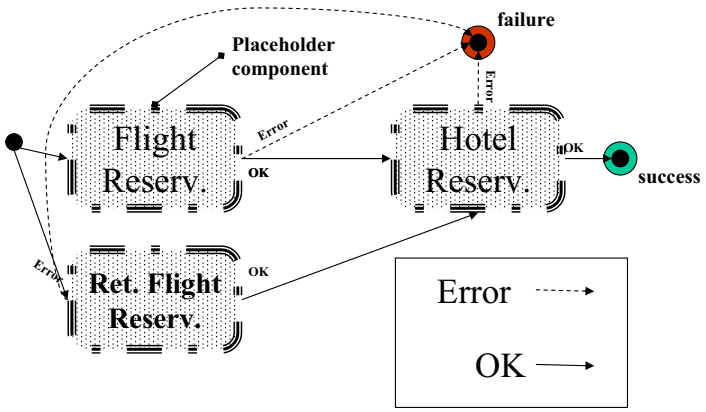


Figure 2. A partially specified workflow graph, describing the application of Figure 1 at the highest level possible.

The place-holder component, under this model, can be considered as a *query* submitted to the component search module, in order to obtain a list of (possibly) relevant components.

Obviously, the place-holder specifications can be as simple as specifying only some keywords related to *non-functional* characteristics of the component (e.g. its description in natural language), but it can soon become complex if we also include *functional* information. For example, the 'Flight Reservation' component can be searched through a place-holder query based on the keywords: *airplane, reserve, flight, trip, take-off*, but we can also ask for a specific method signature to specify the desired destination and take-off time.

The discovery phase we imagine is made up of two steps. First we try to resolve the place-holder by using the components contained in the local repository. If a suitable component is found locally, then this is promptly returned to the user without searching on remote sites. On the other hand, if it cannot be found locally, a *Query Session* is started. The goal is to retrieve a *ranked list* of components that are *relevant* to the specification given in the place-holder plan graph.

As an example, let us consider the above steps in the development process of the example depicted in Figure 1. In a Grid software development environment a programmer could have sketched the abstract workflow plan graph depicted in Figure 2.

Starting from here, the process would be as follows. First, the developer would look for a flight reservation component matching the place-holder. Let us suppose that such a component is available locally. The search module will automatically return a pointer to it and expand the place-holder with the found component (*binding*). Figure 3 shows the workflow graph as it appears at this point of development. In the picture we can see that the found component has been instantiated twice, for both the outward and return flights. Moreover, note that the matching component is a metacomponent, i.e. it is composed of several (interconnected) components.

Then, the user selects the 'Hotel Reservation' place-holder. Since this is not available in the local repository, a query session is initiated. As for the discovery phase, the search process is composed of

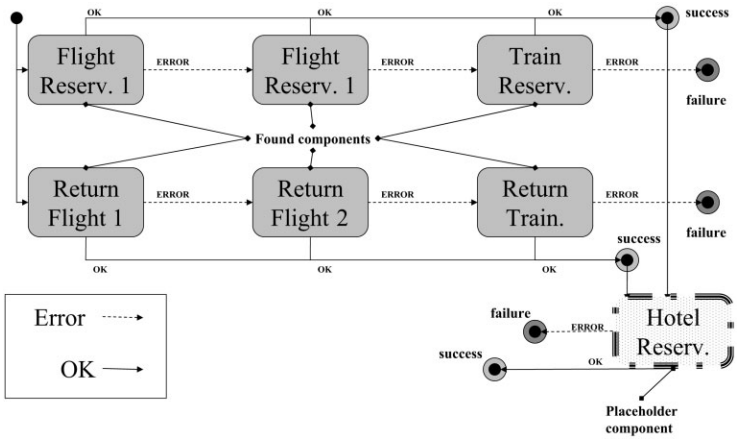


Figure 3. The abstract workflow graph as it appears after the 'flight reservation component' has been found.

two different steps. In the first part, we try to find an initial (possibly inaccurate) list of components. The user then has to refine it until a shorter, and more relevant, list of components is obtained. From this, the user would pick up the most suitable component to replace the corresponding placeholder (*binding*). Finally, when all the components are fully specified, the developer will continue refining the application until it meets his/her original requirements.

The binding phase may be as simple as forwarding the output channel of a component to the input of the next (as for Unix pipes), but it may be more complex if data and/or protocol conversions are needed. In this latter case, a user-driven, framework-assisted procedure is needed. The framework should try to determine the type and the semantics of components' input/output ports, using any available header, XML and textual descriptions, Web ontologies, pattern matching and naming conventions. With this information, the programmer should choose the best chain of conversions, and ask the framework to instantiate an ad hoc filter, performing the transformation needed (for instance, the output of a component needs to be converted from a chain of strings into a Java array of double, and sent over HTTP/SOAP).

If needed, the developer should also be able to describe more complex filtering with some programming language, which should be compiled and deployed with the rest of the application. In our vision, this should be as simple as writing a Perl script to match the input and output of two command-line Unix executables, if we want Grid components to be easily and widely available. This is a field open to research, and we do not want to speculate further on this opportunity here. In the rest of the work, we will assume that components can be made to interact with some user-driven computer-aided effort.

The goal of the search engine module is clear, but what are the possible techniques that can be used to reach this goal?

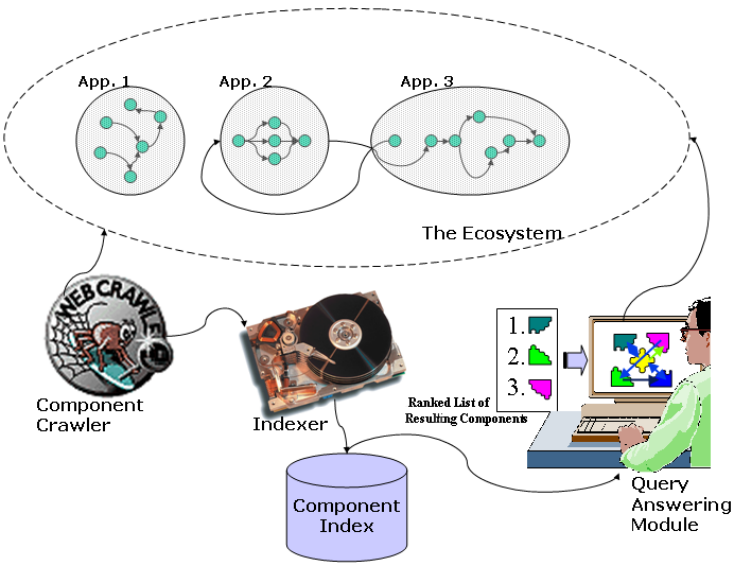


Figure 4. The architecture of GRIDLE.

4.2. GRIDLE: the search module

In the last few years, study of the Web search engine has become a new and important research topic. In particular, several researchers' efforts have been spent on Web models suitable for ranking results of a query to a search engine [20].

We would like to approach the problem of searching software components using this mature technology. We would like to exploit the concept of an *ecosystem* of components to design a solution able to *discover* and *index* applications' building blocks, and allowing the search for the most relevant components for a given query.

Figure 4 shows the overall architecture of our component search engine called *GRIDLE*: *Google*^{TM}*-like Ranking, Indexing and Discovery service for a Link-based Eco-system of software components*. The main modules are the *Component Crawler*, the *Indexer* and the *Query Analyzer*.

The *Component Crawler* module is responsible for automatically retrieving new components: (1) a pre-compiled list of repositories, and (2) an incremental list of repositories continuously updated on the basis of the linking information contained within the crawled metacomponents. Consider that, in our vision, these metacomponents are workflows referring to other lower-level components provided by other repositories. Note that our *ecosystem of components* lives and evolves due to the links present among components. Obviously, in order to make these links significant, we must define how a component can be referred (i.e. Unique Identifier, URL, URI, etc.).

The next module is the *Indexer*, whose job is to build the *index* data structure of GRIDLE. This step is very important, because some information about the relevance of the components within the ecosystem must be discovered and stored in the index. In other words, we have to choose the kind of characteristics we consider to be important for the search purposes, such as:

- functional information: interfaces and running environment requirements;
- non-functional information: textual description, performance and billing;
- linking information.

The *functional information* refers to a more complex but more precise description of the functionality offered by the component. For example, it may declare the number of published methods, their names, their signatures, and so on. The result of the introspection operation is applied to a software component. Furthermore, there are other functional features that we can mention: for example, the specification of the requirements of a component (e.g. *Contracts* [21]).

Non-functional information consists of a textual description, in natural language, of the main functionality of a component. Also, cost and performance information can be described here, along with details about guaranteed QoS, licenses that are needed and so on.

The third characteristic is the information regarding the context where a component can be found. In particular, the *interlinked structure* of metacomponents (workflows) allows for designing smart Ranking algorithms for software components. In particular, we would like to consider the number of in-links of a given component[‡‡] as a factor of high quality (high rank). In addition, we would like to reinforce the quality of a metacomponent on the basis of the importance of the referred components. This last aspect can be better explained using the example graph of Figure 1.

Let us consider the module 'Hotel Reservation 3'. In the ranking phase of the search process, it will obtain an importance coefficient depending on (1) the importance of the 'Flight Reservation' component, and (2) the importance of the 'Car' component. The former is used to compute the rank by using a modified version of the well-known PageRank iterative algorithm. Then, the importance derived by (2) is computed by increasing the weight of 'Hotel Reservation 3' by a constant factor α proportional to the relevance of 'Car'.

The last module of GRIDLE is *Query Answering*, which actually resolves the queries on the basis of the index. The main function carried out by this module is to return a list containing the most relevant components in response to a user's query. The relevance score is computed by the *ranking module* using the aspects outlined above as comparison metrics. In principle, the algorithm is simple:

- for each component C previously indexed:
  - — compute the distance among C and the place-holder query Q;
  - — place the score along with C in a list L;
- select from L the r most relevant component with respect to Q.

In this algorithm, the main interesting point is the computation of the distance. In our model, this should come out from a mix of the four relevance judgements given by the above-mentioned criteria. Then, the results set L is presented to the user for further refinements.

[‡‡] An *in-link* is a directed link which targets the current component.

4.3. Our prototype

In our first few experiments, we collected and analyzed a large set of components in the form of Java classes. Clearly, Java classes are a very simplified model of software components, because they are supported by only one language, they cannot cooperate with software developed, using other languages, but they also support some very important features including the following: their interface can be easily manipulated by introspection; they are easily distributed in the form of single JAR files; they have a very consistent documentation in the form of JavaDocs; and they can be manipulated visually in some IDEs (BeanBox, BeanBuilder etc.).

We collected information about approximately 50 000 Java classes*. We were able to retrieve very high-quality Java Docs for the Java API (Java 1.4.2 API and Java 1.5.0 API), and for some very popular projects including Apache Struts; Globus 3.9.3 MDS; Globus 3.9.3 Core and Tools; Tomcat Catalina; JXTA; Apache Lucene; Apache Tomcat 4.0; DBXML; ANT; and Nutch.

For each class, we determined which other classes it used and which other classes used it. With usage, we mean the fact that a class A has methods returning, or using as an argument, objects of another class B: in this case, we record a link from A to B. This way, we generate a directed graph describing the social network of the Java library.

Out of our preliminary results, we developed a very simple search engine, able to find high-relevance classes out of our repository. Classes can be ranked by TF.IDF (a common information retrieval method, based on a metric that takes into account both the number of occurrences of a term within each document and the number of documents in which the term itself appears) or by Class Rank, our version of PageRank for Java classes, based on the class usage links. Figure 5 shows the first Web interface of our tool.

Class Rank is a very simple algorithm that builds on the popular PageRank algorithm used in Google [15]. To determine the rank of a class C, we iterate the following formula:

$$rank_C = \lambda + (1 - \lambda) \sum_{i \in inlinks_C} \frac{rank_i}{\#outlinks_i}$$

where $inlinks_C$ is the set of classes that use C (with a link into C), $\#outlinks_i$ is the number of classes used by i (the number of links out of i), and λ is a small factor, usually around 0.15.

We were able to observe some very interesting results. The highest-ranking classes are clearly some basic Java API classes, such as `String`, `Object` and `Exception`. Nonetheless, classes from other projects are apparently very popular among developers: #7 is Apache `MessageResources`, #11 is Tomcat `CharChunk`, #14 is DBXML `Value` and #73 is JXTA `ID`. These classes are very general, and are used by developers of unrelated applications.

We could verify that in many cases Class Rank is more relevant than TD.IDF, especially when the class name is not textually similar to the function being searched for. For instance, if the developer is trying to write data to a file, and performs a query such as '*file writer*', TF.IDF will return, in order:

1. javax.jnlp.JNLPRandomAccessFile, from JNLP API Reference 1.5;
2. javax.swing.filechooser.FileSystemView, from Java 2 Platform SE 5.0;

*To be precise, 49 425 different classes.

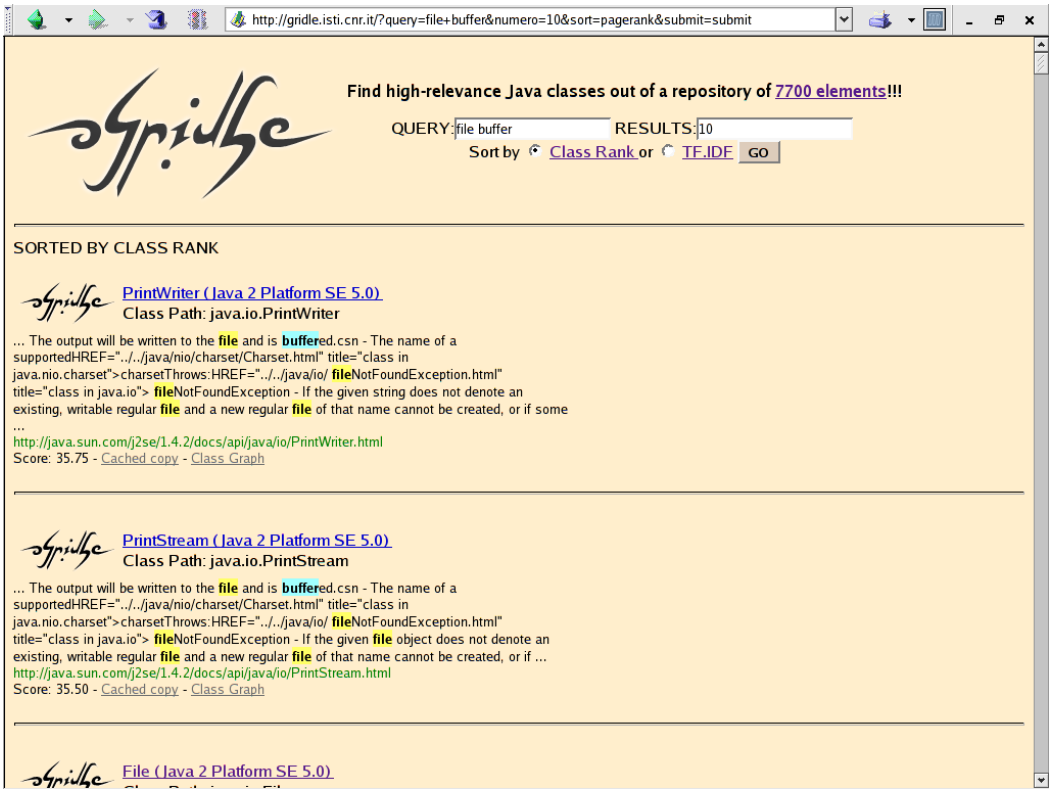


Figure 5. Web-interface of .

3. java.io.FileOutputStream, from Java 2 Platform SE 5.0;
4. java.io.RandomAccessFile, from Java 2 Platform SE 5.0.

The second class is clearly unrelated to the problem under analysis, and only the third class is probably what the user is looking for. On the other hand, Class Rank will return four classes from the Java API (Java 2 Platform SE 5.0):

1. java.io.PrintWriter;
2. java.io.PrintStream;
3. java.io.File;
4. java.util.Formatter;

which are all probably better matches.

5. CONCLUSIONS

In this contribution, we presented our vision of a new tool allowing the design of workflow-based Grid applications where a composition of different workflows can be seen as a single autonomous meta-component. The main issue presented in the work is the *component search service*, which allows users to locate the components they need. We believe that in the near future there will be a growing demand for ready-made software services, and current Web search technologies will help in the deployment of effective solutions. The search engine in our opinion should be able to *rank* components on the basis of their similarity with the place-holder description, their popularity among developers (something similar to the hit count), their use within other services (similarly to PageRank) etc.

Clearly, it is of primary importance that there exists a quick, efficient, automatic way to deploy software components out of existing code. In our opinion, there is the need for automatic tools able to extract signature information from legacy code, and able to create the bridging code needed to make the component communicate with other entities, designed with different languages or running on different platforms.

When all these services become available, building a Grid application will become a straightforward process. A non-expert user, aided by a graphical environment, will give a high-level description of the desired operations, which will be found, and possibly paid for, out of a quickly evolving market of services. At that point, the whole Grid will become like a virtual machine, tapping the power of a vast number of resources.

REFERENCES

1. Armstrong R, Gannon D, Geist A, Keahey K, Kohn S, McInnes L, Parker S, Smolinski B. Toward a common component architecture for high-performance scientific computing. *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press: Los Alamitos, CA, 1999; 13.
2. Zhuge H, Shi X. Toward the Eco-Grid: A harmoniously evolved interconnection environment. *Communications of the ACM* 2004; **47**:78–83.
3. The Open Directory Project. http://www.dmoz.org [12 July 2005].
4. Page L, Brin S, Motwani R, Winograd T. The pagerank citation ranking: Bringing order to the Web. *Technical Report*, Stanford Digital Library Technologies Project, 1998.
5. Szyperski C, Pfister C. WCOP '96 Workshop Report. *ECOOP '96 Workshop Reader* (*Lecture Notes in Computer Science*, vol. 1038). Springer: Berlin, 1996.
6. Business process execution language for Web services. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf [12 July 2005].
7. Oinn T, Addis M, Ferris J, Marvin D, Greenwood M, Carver T, Wipat A, Li P. Taverna, lessons in creating a workflow environment for the life sciences. *Proceedings of the GGF 10 Workflow Workshop*, Berlin, 2004.
8. Workflow in Grid systems. http://www.extreme.indiana.edu/groc/ggf10-ww/ [12 July 2005].
9. Braga RMM, Werner CML, Mattoso M. Odysseysearch: An agent system for component. *Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, Portland, OR, May 2003.
10. Seacord RC, Hissam SA, Wallnau KC. Agora: A search engine for software components. *Technical Report ESC-TR-98-011*, Carnegie Mellon—Software Engineering Institute, Pittsburgh, PA, 1998.
11. Frakes WB, Pole TP. An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering* 1994; **20**(8):617–630.
12. Inoue K, Yokomori R, Fujiwara H, Yamamoto T, Matsushita M, Kusumoto S. Component rank: Relative significance rank for software component search. *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003. IEEE Computer Society Press, Los Alamitos, CA, 2003; 14–24.
13. Cannataro M, Congiusta A, Pugliese A, Talia D, Trunfio P. Distributed data mining on Grids: Services, tools, and applications. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics* 2004; **34**:2451–2465.

14. Lord P, Wroe C, Stevens R, Goble C, Miles S, Moreau L, Decker K, Payne T, Papay J. Semantic and personalised service discovery. *Proceedings of the Workshop on Knowledge Grid and Grid Intelligence (KGGI'03)*, in conjunction with *2003 IEEE/WIC International Conference on Web Intelligence/Intelligent Agent Technology*, 2003.
15. Brin S, Page L. The anatomy of a large-scale hypertextual Web search engine. *Proceedings of the 7th World Wide Web Conference on Computer Networks*, vols. 1–7, April 1998; 107–117.
16. Van Rijsbergen CJ. *Information Retrieval*. Butterworths: London, 1979. Available at: http://www.dcs.gla.ac.uk/Keith/Preface.html [12 July 2005].
17. Yang C-SD, Pollock LL. All-uses testing of shared memory parallel programs. *Software Testing, Verification, and Reliability Journal* 2003; **13**(1):3–24.
18. Jackson N. Pyglobus: A python interface to the globus toolkit. *Concurrency and Computation: Practice and Experience* 2002; **14**(13–15):1075–1084.
19. Thain D, Tannenbaum T, Livny M. Condor and the Grid. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley: New York, 2003; 299–335.
20. Baeza-Yates RA, Ribeiro-Neto BA. *Modern Information Retrieval*. Addison-Wesley–Longmann: Reading, MA, 1999.
21. Holland IM. Specifying reusable components using contracts. *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, vol. 615, Madsen OL (ed.). Springer: Berlin, 1992; 287–308.

