Toward GRIDLE: A Way to Build Grid Applications Searching Through an Ecosystem of Components

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

[1] Institute for Information Science and Technologies
ISTI - CNR, Pisa, Italy
{diego.puppin, fabrizio.silvestri, domenico.laforenza}@isti.cnr.it
[2] Università di Venezia, Ca' Foscari
Venezia, Italy
orlando@dsi.unive.it

1 Introduction

Today, the development of Grid applications is considered a nightmare, due to lack of Grid programming environments, standards, off-the-shelf software components, etc. Nonetheless, several researchers believe that economic principles will guide the future development of the Grid: an open market of services and resources will become available to developers, who will choose to use computing time and software solutions from different vendors, sold at different prices, with different performance and QoS.

Standardization efforts on component models, integration platforms, and business domain concepts based on XML will accelerate the usage and spreading of blocks for building component-based Grid services and Grid applications. We can expect that, in a very near future, there will be thousands of open-market components available on the Grid. Grid programming will consist of selecting, coordinating and deploying *components* chosen from this large software market: the problem will be to find the best component that fits the requirements and with the best performance/price trade-off.

Two essential requirements for a software market-place have been slow to emerge: (1) standard, interchangeable *components*, orchestrated according to complex job *workflows*, and (2) the consumers' ability to *search* and *retrieve* the right parts for the job at hand. While we can register several activities in the Grid community directed toward the definition of a standard for high performance components, the effort to develop searching and retrieving services for software Grid components has been limited.

In this contribution, we will discuss the preliminary design issues of GRIDLE, a Grid component search service, which will follow the direction of using the mature technology of Web search engines to discover software components on the Grid. GRIDLE, a *Google^{TM}-like Ranking, Indexing and Discovery service for a Link-based Eco-system of software components*, is thought as the key tool in a more modern and natural framework for application development.

Nowadays, a typical process of application development may look as follows:

> the programming environment offers a repository (*palette*) of possible components, from which developers can choose the most appropriate ones for their purpose. These components are locally developed, or they are remote, but explicitly and perfectly known: the programmer, or the environment, knows the availability of trusted components, their location in the Grid, their interfaces and their behavior. The developer can build complex workflows, according to which components can be deployed, and their ports can be interconnected appropriately.

We envision a different approach:

> a Grid programmer needs to fulfill a complex job. S/he starts the application development by specifying a *high-level workflow plan*: for each needed component, s/he specifies functional and not functional requirements, which are used to query the search service. GRIDLE will start a search session and will present the user a ranked list of components, which are partially/totally matching the requirements. The programmer is then in charge of adapting interfaces or making the component compatible with the run-time system supported by her/his computing environment. Using a suitable cost and performance model, and the information offered by the components themselves, the framework tries to predict the overall cost of the application, and its expected performance. It should also verify, on behalf of the developer, the credentials of the chosen components (with suitable security technology). With this information, the user can be driven to a cost-effective solution.

In other words, GRIDLE will work like a Web search engine, i.e. offering a simple interface to a large, evolving market of components. As it will be shown in the next sections, a user should be given the opportunity to find the needed software components simply through a short description of his/her requirements.

Looking at the evolution of search engine technology, which will constitute the basis for the design of GRIDLE, we can observe the increasing importance of the adoption of new ranking methods for pages, besides the well-known Information Retrieval models.

In particular, search engines make use of the link structure of the Web to calculate a quality ranking for each Web page. For example, Google^{TM} interprets a link to a given page as a vote. These votes produces a ranking method called PageRank, which can be thought as an objective measure of a web page's importance that corresponds well with people's subjective idea of importance. In other terms, PageRank works because we can imagine the Web as an *ecosystem* of hyper-linked data contents, where such contents *cooperate* – i.e. they are hyper-linked to each other in order to offer more complete information – and *compete* – i.e. they publish information and try to make themselves more visible than others. From this Darwinian process, only the most important pages become accessible through a Web search engine. Note that, similarly to a real

ecosystem, the winners in this competing environment will be those that will be chosen as a partner (linked to) by a large number of other information providers (Web pages).

We believe that a similar process will happen for Grid components and their ecosystem. They will *cooperate* – i.e. they will be composed by means of specific workflows – and *compete* – i.e. they will try to make themselves more visible than others in the large component marketplace. So, we envision that workflows will play a role similar to hyperlinked pages in the Web, i.e. as a means to improve the visibility of available components on the Grid. In practice, workflows can be considered as particular documents with hyperlinks to the components involved: GRIDLE could exploit them to improve the precision of a search. We also believe that Web pages describing components and related features will be also made available, and thus the more traditional Web search engine methods will continue being adopted in combination with this new technology.

This contribution is organized as follows. In the next section, we describe emerging issues related to the creation and publication of software components, with special attention to the wrapping of existing code: we compare several component-oriented frameworks, and we highlight some key features. Then, we comment on some standards of application description. Starting from these, in section 4, we give our vision of a new way to build applications starting from software components available on the Grid. We then conclude, summarizing the most challenging research problems, which we plan to address in the future.

2 Component-based Grid Programming

When Unix introduced pipes in 1972 [1], new opportunities for software engineering became available: nobody before thought seriously about building *software solutions* out of existing tools, and designing them as *filters* so that they could be easily used as building blocks for larger tasks: currently, all Unix programmers design chains of pipes for their needs, instead of writing new applications from scratch, as part of their daily routine.

Unix filters and the pipe composition model represent the first successful example of a component-based design. It offers several important features, among which *substitutability*, i.e. the pieces building up a complete application are designed to be added, removed and replaced with cheaper, faster, more robust ones; *customization*, because through some user interface (e.g. the Unix command line) the components offer a certain degree of configurability; *heterogeneity*, since tools developed with different languages can interoperate through a well-known, common interface (Unix text stream).

In literature, "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" [2]. In a growing Grid market, the way software components are defined and modeled has a strong impact on the speed of a transition to a component-oriented application design. In this section, we describe the state of the art of software component

design, describing some existing component frameworks. Then, we consider pros and cons of the existing approaches, with some quantitative analysis. Next, we highlight our proposal of a new approach to component modeling and linking, with a brief overview of tools for automatic wrapping.

2.1 Component-oriented Frameworks

Here, we analyze the state of the art of software component design. In particular, we focus on two frameworks that are representative, in our opinion, of two opposite approaches to component modeling: CCA [3] and Java Beans [4]. CCA is based on the use of an Interface Definition Language (IDL). Java Beans is based on Java's most modern features (introspection, object-oriented design...) to provide a simple and portable programming environment.

The Common Component Architecture (CCA). CCA [3] is a proposed standard aimed at promoting the integration of heterogenous software components, by defining a minimal set of required services that every component should feature: *Port, getPort, setServices.*

In the CCA vision, a component is a software entity that allows different languages, different hardware/software architectures to co-operate. It supports a structured, modular approach to software development, very useful in the case of third-party libraries and external services. A component can be built out of a library or a class (object type), but it can exist only with the support of a framework, responsible for connecting the interfaces of the different components, in order to build a complete application.

Every component exports one or more input and output ports (*uses* and *provides* ports), identified by their type. Inheritance and specialization allows the programmer to refine definition and semantics of a component. Also, multiple ports can offer a different point of view of the same service.

CCA is implemented by a variety of frameworks, offering different programming models: CCAffeine is targeted to SPMD (Single Program, Multiple Data) parallel applications, based on the connection of locally available components [5]; CCAT/XCAT is more suitable to Grid/distributed applications, as it can connects component located on remote machines (e.g. in a Globus-based Grid) [6]; SCIRun supports parallel multi-threaded application, based on local connection [7]; finally, Decaf is aimed at offering language interoperability for a sequential application.

In the next paragraphs, we will discuss in detail CCAffeine and Babel, a tool used by CCAffeine and other CCA framework to support language heterogeneity.

The CCAffeine Framework. CCAffeine [5] is a CCA-compliant framework oriented to SPMD (or SCMD — Single Component Multiple Data) applications. It can assemble together heterogeneous, internally parallel components. As an example, in Figure 1, three parallel components, each composed of 5 parallel processes, build up an application. Intra-processor communication, among different

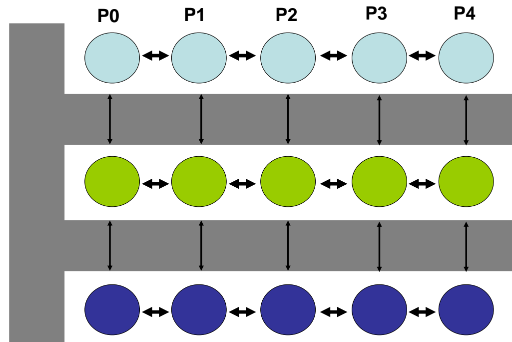

Figure1. Communications among and within components. Five machines (P0 through P4) are running an application composed of three parallel components.

components (vertical arrows) is managed by the framework: it is responsible for data conversion if the components are heterogeneous. On the other side, interprocessor communication, within a parallel component (horizontal arrows), is responsibility of the component developer, who can use the tool of his/her choice (MPI, PVM, RPC, shared memory or any other communication library).

CCAffeine offers a basic GUI (see Figure 2) to deploy and connect components: a *palette* listing the available components is shown to the user, who can pick up the needed functions from it. Using a simple drag-and-drop interface, s/he can connect them into a graph that describes the complete application. The framework is responsible for instantiating a copy of the application graph on every machine in the computing environment. CCAffeine internally uses MPI and creates the requested components on all machine in the MPI world. The *run* command will start, in parallel, execution on each machine.

Over all, CCAffeine offers a language-independent wrapping to parallel components (mostly, MPI-based). Thus, it supports the usage of heterogenous development environments (SO shared-object libraries can be easily connected), leveraging the services of Babel.

Babel. Babel [8] is a tool suite featuring an interface-description language oriented to high-performance scientific programming. It is aimed at supporting the co-operation of software written with different languages. It offers a set of functionalities similar to CORBA [9], but enriched by scientific features, among which support for dynamic multi-dimensional arrays, complex numbers, in-process optimizations, support to C, C++, Fortran-77, Java, Python, Fortran-90.

The usage of Babel SIDL is similar to that of other IDL-based tools. Given the interface description, the Babel parser creates: *IOR files* with the Internal Object Representation (IOR) of the needed data structures; *Skel files*, responsible for the conversion of the IOR to the native data representation; *Impl files*, collecting

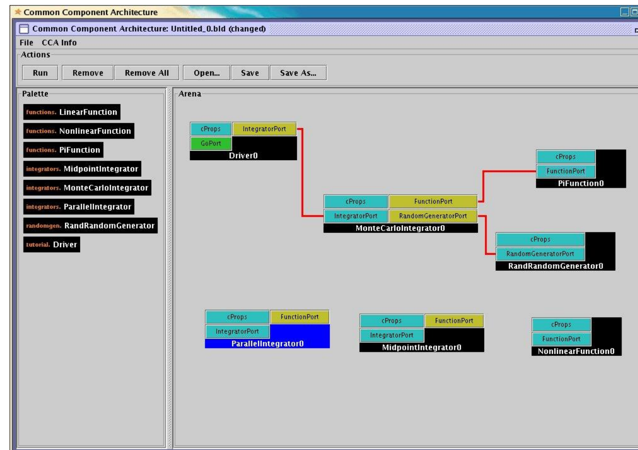

Figure2. CCAffeine graphical interface.

the implementation, in any supported language; and *Stub files*, needed to connect to the Babel support libraries.

Babel creates server- and client-side support: it exports a set of stubs to be used to call the implemented object from a program written in any other supported language. In our experience, Babel does not seem to use effective shortcuts when implementing communication among homogeneous components: we observed that two C++ components need to go through the slower run-time support of abstracted data, which is needed by heterogeneous components, instead of using the native format. In Table 1, we show the call stack for a simple program, completely written in C++ (thus, no heterogeneity is present). The call of *evaluate()* by *integrate()* is mediated by several layers of framework support.

This is a problem common to other IDLs: their run-time support does not provide effective shortcuts to use native data structures, and the language-independent data abstraction needs to be used, with an evident performance overhead.

The process of creating a CCA component, in particular a CCAffeine component, from the Babel interface requires a complex series of steps:

1. definition of component interfaces using Babel;
2. implementation of component algorithms in the Babel-generated Impl files (in any supported language);
3. definition of CCA ports in the Impl files;
4. creation of a wrapper and a CCA definition file;
5. creation of a suitable makefile.

In our experience, this process could cause problems of software engineering and is potentially error-prone. In particular, there are some naming conventions

```
#0 functions::PiFunction_impl::evaluate() at functions_PiFunction_Impl.cc:47
#1 skel_functions_PiFunction_evaluate () at functions_PiFunction_Skel.cc:32
#2 functions::Function::evaluate() at functions_Function.cc:236
#3 integrators::MonteCarloIntegrator_impl::integrate() at
   integrators_MonteCarloIntegrator_Impl.cc:134
#4 skel_integrators_MonteCarloIntegrator_integrate () at
   integrators_MonteCarloIntegrator_Skel.cc:48
#5 integrators::Integrator::integrate() at integrators_Integrator.cc:237
#6 tutorial::Driver_impl::go() at tutorial_Driver_Impl.cc:100
#7 skel_tutorial_Driver_go () at tutorial_Driver_Skel.cc:45
#8 gov::cca::ports::GoPort::go() at gov_cca_ports_GoPort.cc:244
#9 BabelOpaqueGoPort::go() at BabelOpaqueGoPort.cc:28
#10 ConnectionFramework::goOne() at ConnectionFramework.cxx:1139
#11 DefaultBuilderModel::goOne() at DefaultBuilderModel.cxx:222
#12 CmdActionCCAGo::doIt() at CmdActionCCAGo.cxx:55
#13 CmdParse::parse() at CmdParse.cxx:564
#14 CmdLineBuilderController2::parse() at CmdLineBuilderController2.cxx:118
#15 CmdLineClient::main() at CmdLineClient.cxx:861
#16 main () at CmdLineClientMain.cxx:318
#17 __libc_start_main () from /lib/tls/libc.so.6
```

Table1. The call stack for a simple program. The *go()* method (line #9) activates the *integrate()* function (#5), which calls then *evaluate()* (#0).

that needs to be respected in the development of the wrapper and CCA files (fourth step), which can be missed and are not recognized properly by the development tools.

Performance. As said, the main goal of CCAffeine is to improve the interoperability of software components. High performance is responsibility of the application developer, who has to tune computation, communication and data placement to this goal.

In our tests, we observed huge performance overhead when homogeneous components were communicating. In Figure 3, a comparison of a C++ implementation vs. a component implementation of a simple integration program is shown. A parallel MPI integrator distributes function evaluations among a set of workers. Partial results are then *reduced* to compute the final sum. The data flow for the CCAffeine implementation is: a driver component invokes the services of a generic parallel integrator, which invokes the method *evaluate()* of a function component. The parallel integrator is responsible for orchestrating the final reduction. In the C++ version, components are replaced by functions, and service invocations are replaced by function calls. A second version was tested, where the function was inlined in the integrator body. When the function to be integrated is an autonomous component, a large overhead for the function call can be observed. When the function is inlined, the framework start-up time dominates.

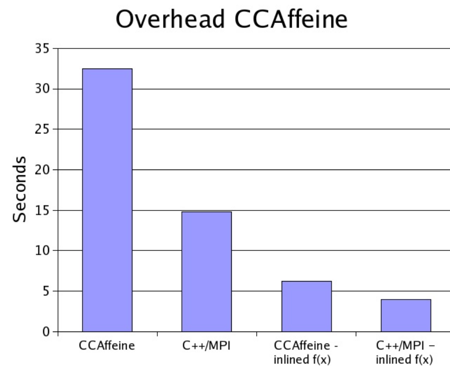

Figure3. Comparison of CCAffeine and native code. The first column is CCAffeine implementation; the second is C++/MPI; the third is CCAffeine with inlined function; the fourth is C++/MPI with inlined function.

Tests were performed on a cluster of eight dual-Pentium machines, running Linux RedHat, Babel 0.8.4, and CCAffeine 0.4.1.

The overall performance of CCAffeine is strongly affected by the features of the underlying run-time support of Babel. We want here to highlight an important feature of Babel arrays. As well known, an important factor affecting performance when manipulating matrices is the ordering of column and rows. If the order is correct, elements can be accessed sequentially, otherwise a more complex access is needed. Babel SIDL allows the programmer to define the order to be used by the data passed to a component: a component can choose to access row-major or column-major matrices. In row-major mode, rows can be scanned sequentially, increasing the column number. We tested the performance of a simple matrix multiplication, with different matrix ordering. In Table 2, one can compare the costs for some choices of orders for the matrices A, B and C (with $C = A \times B$). R stands for row-major, and C for column-major.

The last column indicates what the cost is when a matrix is transposed because it does not match the parameter definition: if, at the moment of call, the data in the actual parameter are ordered differently from what is expected, the system is responsible for transposing it, a costly operation for large matrices. This operation has a high cost, and the programmer is not warned about this.

Also, we have to emphasize that components have to use the Babel SIDL array data abstraction, and there is no automatic optimization: if the matrices are defined with a sub-optimal ordering, the code will have bad performance, as it can be seen in the second and third columns, where the matrix ordering does not match the natural structure of the multiplication algorithm.

According to the developers, component overhead is “negligible [...] for component implementation and abstract interfaces when using appropriate levels of abstraction.” Nonetheless, they also measure that parameter passing can be

Size	RCR	CRR	CRC	Transpose
16	0.06	0.02	0.02	0.00
32	0.08	0.03	0.03	0.00
64	0.12	0.1	0.09	0.10
128	0.37	0.31	0.29	0.31
256	1.62	2.99	2.65	2.30
512	12.11	54.95	54.13	62.32

Table2. Matrix multiply, with different orderings. Time in seconds.

three times slower than native code [10]. More data to compare this framework with native code are given in section 2.2.

Java Beans. Java Beans API [4] is a mature software standard developed by a consortium, led by SUN, with the goals of designing a portable, simple, high-quality API for component software development. Concretely, a Java Bean is a reusable software component that can be manipulated visually in a builder tool. While visual manipulation is an important part of software development, and many beans have important GUI functionalities (scrolling bars, editing windows...), beans can be more complex and can be used to model larger software entities: data-bases, observational instrumentations, and visualization facilities can be modeled as beans and integrated into the development framework.

The most advanced and useful features of beans are introspection, customization, and the event-driven programming model.

Introspection is a very powerful tool for component publication and connection: Java classes are able to analyze their own structure, and publish their interface. This is present in a very basic form in any Java class, which can dump its own identity to a string, when asked to perform the method `toString()`. Also, `Serializable` classes are able to store their state to a stream.

Java Beans offer a more advanced set of information methods, able to retrieve the name of the relevant methods, the set of fired events, the interesting properties. If desided, an explicit `BeanInfo` service can be provided by the Bean, e.g. if the internal properties of the Bean are to be hidden or abstracted. Also, a *design-time customizer* can be offered with the Bean. This tool will be invoked by the design tool, instead of the standard configuration interface.

Java Beans are mostly event-driven. This means that they can react asynchronously to particular circumstances that happen during application execution, for example clock ticks or user input. More advanced message-driven beans are also possible, using the run-time support offered by `Java 2 Enterprise Edition` (J2EE) [11] and similar software packages. Moreover, Beans can be multi-threaded, and can operate on multiple requests at a time.

Design Time and Run Time. Java Beans recognize two important moments in the application development. *Design time* is the moment when the bean is manipulated, customized and composed into an application. Customization tools

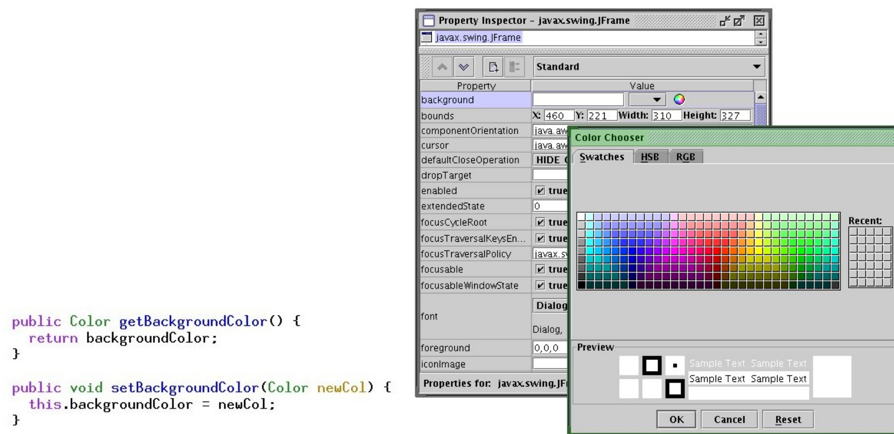

Figure4. Example of design convention: when a pair of *get-set* methods are found, the framework can identify a property and offer the correct customization tool.

offered by the bean (called `Customizer`) are invoked at design time by the design framework. When the application is completed, and it is packaged to run (*run time*), the design-time code is stripped out, and only the run-time code is kept in the application code. `Customizer`s are available for complex beans that require long, detailed customization, or when the internal implementation needs to be abstracted or hidden.

This strategy can be very good, in general, for complex component: versatile configuration tools can be packaged along with the main code, and then stripped out to reach performance at run-time.

Design Conventions. The Java standard is highly based on naming and coding conventions, usually called, in Java Beans terminology, *design patterns* [4], which simplify and coordinate the interaction of software components: these standards are highly suggested (but not mandatory) and are exploited at the moment of interaction. Java Beans frameworks look for these naming conventions in the class signature, and use them to extract some semantic information.

For example, when two methods, `PropertyType getSomeProperty()` and `void setSomeProperty(PropertyType s)` are found, a property `someProperty` of type `PropertyType` is defined, and presented to the user at design time.

This set of standards allow automatic tools to present suitable customizing tools for the object's properties. For instance, `BeanBox` and `BeanBuilder` will present a palette for a color attribute, a list of choice for an enumeration type and so on (see Figure 4).

Other Component-based Environments. The *CORBA Component Model* [9] is an extension of CORBA with the goal of defining component interfaces, and

objects with persistent state. CORBA is a powerful tool to perform method invocation on objects running on remote servers, based on an IDL. PARDIS [12] extends CORBA in another direction, so to allow sequential and parallel clients (SPMD and MPMD) to interact with sequential and parallel servers. With similar goal, further results have been reached by PAWS [13] and Ligature [14].

Web Services are another standard to define and represent software components. The OGSI (Open Grid Service Infrastructure) forum has enriched the standard to include some more Grid-oriented services [15]. The design of a Web service is quite straightforward, and highly automated starting from Java classes. Tools are also available for other languages (mostly, C++).

Another emerging framework is Microsoft .NET [16], which presents an integrated model of component-oriented software and Web-based services. We will consider its features in the comparison presented below.

2.2 A New Approach. What Is Needed.

We want, here, to summarize pros and cons of the discussed environments, in order to define the requirement of a new approach to software component design. First, we give a qualitative comparison of the critical features of the two main approaches (CCA and Java Beans) with some considerations about solutions offered by .NET. Then, we compare their performance, with particular attention to the overhead introduced by Babel IDL.

Qualitative Comparison The two approaches present strong differences in the way they face many issues related to software development.

Re-use of legacy code. In our experience, the process of modifying existing code in order to use the IDL-generated skeletons and stubs is complex and potentially error-prone. An approach following the example of Java, where any class can be quickly transformed into a Bean and then plugged into the framework, is to be preferred.

Component publication. With Java Beans, there is no need to define interfaces and to pass them around. The `JAR` file, storing the code for the component, can be queried: the object is able to introspect and to publish its own methods and data. The CCA Babel interface, instead, needs to be passed around as part of the code documentation, with strong problem of component packaging.

Architecture and language portability. Java byte code is designed to run on any platform that offers a Java virtual machine: this includes Sparc, Macintosh, Linux, Windows, and many portable devices. In the context of Grid computing, a component can be moved to and executed on different resources at every run: Java is a very simple way to ensure that the code is portable.

On the other side, components developed with Babel/CCA are expected to run natively, possibly after re-compiling, on any CCA-compliant framework. Also, heterogeneous components can inter-operate.

.NET offers an approach similar to Java, using a virtual machine for Windows system, called Common Language Run-time (CLR), which has been partially ported to Linux by the Mono project team [17]. Also, it can compile a variety of languages to run on the virtual machine, thus allowing heterogeneous components to operate easily together.

Visual manipulation. Visual manipulation can be an important part of a component-oriented framework if used by non-expert programmers: this is an interesting feature if the component is to be used in a Problem Solving Environment (PSE) or some computer-aided application designer.

Java Beans and CCA components are both designed to be easily manipulated with graphical tools. Nonetheless, Java Beans offer a more versatile interface (i.e. a specific `Customizer` for complex Beans) which inherits the flexibility of Java.

Standardization. Java is one of the most wide-spread software standards. There is large availability of software, and a large number of discussion symposia. CCA is still a proposal of standard, which is competing with several others for large acceptance. .NET is another emerging standard, pushed strongly by Microsoft, but also accepted in the open software community for its advanced features.

Type control. CCA offers a very strong type control, based on port typing and object inheritance/specialization. In Java, type matching can be sometimes overridden if methods with no arguments are used: with Java Beans, control on the type of communicating entities is limited. .NET, instead, offers a strong typing system, common to all supported languages, known as Common Type System (CTS). Strong control is needed if the component is to be used by a wide public.

Performance of numeric applications. Java code running on a virtual machine can have lower performance than optimized C or Fortran code. This is due to the overhead introduced by the virtual machine, to limits of the compiling process of Java, and to the lack of high-performance numerical types. On the other side, CCAffeine components incurs in a high overhead due to type conversion and data abstraction. .NET has problems similar to Java, even if its virtual machine, more modern, is designed to emphasize the advantages of just-in-time compiling.

Also, most of scientific/numeric code is written in languages other than Java (Fortran, C): it can be wrapped to run within CCA with some effort, while porting to Java can be much more costly, even if some tools are available to this goal.

Quantitative Comparison. The scientific community is usually hostile to the use of Java for high-performance computing. The main critique goes to the inefficiency of its most common implementation, based on byte code and a virtual machine for its execution.

Nonetheless, great advancements have been made in recent years: frequent forums are organized among research groups to share results about high-performance Java applications, including the Java Grande forum [18]. Among other

language	files	lines
Java	1	68
C	1	81
CCAffeine (generated)	202	10000+
(modified)	8	400+
(written)	4	200+

Table3. Number of files and lines for a simple matrix multiplication with different languages and environments. CCAffeine components were implemented using C++ skeletons.

works, Moreira et al. [19] have shown that very high performance (50% to 80% of Fortran performance) can be reached by Java, using suitable classes and compiler optimizations, on numeric kernels. Kielmann et al. [20] were able to execute threads of a Java application, in parallel on different machines, in a very efficient way.

Java is, without any doubt, a tool of choice for the design of portable interfaces, web applications, data-base wrappers and so on. Nonetheless, the low throughput of data-intensive kernels limited the usage of Java in the scientific community, which seems to be more attracted by the use of IDLs to reach interoperability.

Clearly, we do not have to overlook the high overhead imposed by the latter solution: in term of programming, and in term of performance. As shown above, the overhead to the developer is very high when s/he uses a IDL.

To give a more quantitative evaluation of the performance of Java vs. a SIDL-based component, we considered three implementations of a simple matrix multiply. One is a very simple C implementation, using pointers to integer arrays to store the matrices. The second defines and then uses a new Java class, which offers methods for multiplication and printing of matrices. The third uses Babel to define the interface for a CCAffeine component that performs the product of two given matrices, and another that prints a given matrix. The implementation is then written in C++.

The Java implementation is a single file of 68 lines. In C, 81 lines are needed. Babel generates 202 files among stub, skeleton and implementation files. 8 files were modified to incorporate the C++ implementation (adapted from the C implementation to use Babel arrays) and 4 were written from scratch (see Table 3).

The Java implementation is very compact, and can be run on any machine that runs a Java Virtual Machine or a Java compiler. Also, the new class could be easily wrapped as a Java Bean and then used within any Java Bean framework.

On the other side, components developed with Babel and CCAffeine are expected to interoperate with any other software developed with the Babel interface, even if written in another language, and can run on any CCA-compliant framework (XCAT, CCAffeine...), but the cost to get interoperability across different programming languages through Babel is very high, in term of performance and programming complexity. In Figure 5 we compare the CCA imple-

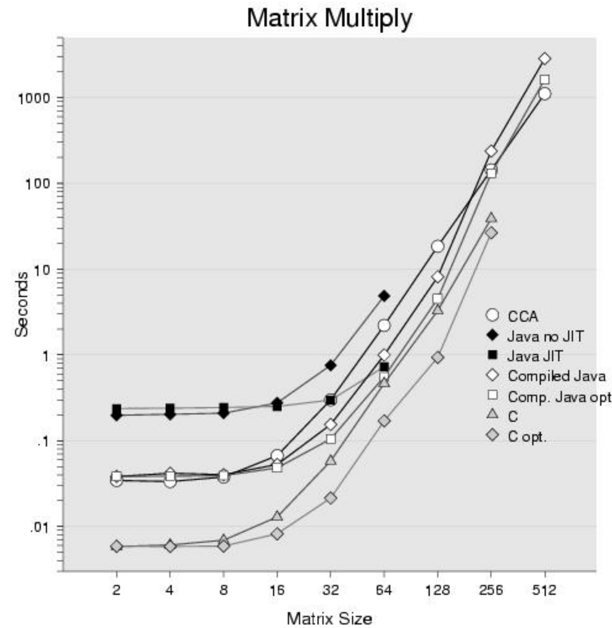

Figure5. Comparison of matrix multiply, with CCAffeine, C, and Java.

mentation, with Java (bytecode, bytecode with Just-In-Time compiler, compiled with *gcj* and compiled with optimizations -*O3*), with C (compiled with and without optimizations).

The overhead introduced by Babel wrapping is very high. In particular, in order to be portable, the implementation of arrays is not native and incurs high run-time performance degradation. In fact, in this test, the overhead registered by the use of Babel is largely due to the use of non-native data types, which is independent by the size of the application: even if framework and start-up overhead are smaller with very large applications, the overhead due to data structures is common to any numerical code using this tool.

The Java code, compiled with optimization, reaches a performance comparable to the CCAffeine implementation. As expected, both fall far behind the faster C implementation.

2.3 A Choice for a Component Model

As shown, two main ways are available today for component publication and management: one is based on a description language, along with compilers to generate stubs and skeletons; the second is based on object introspection and coding conventions, and allows a very fast deployment of Java classes.

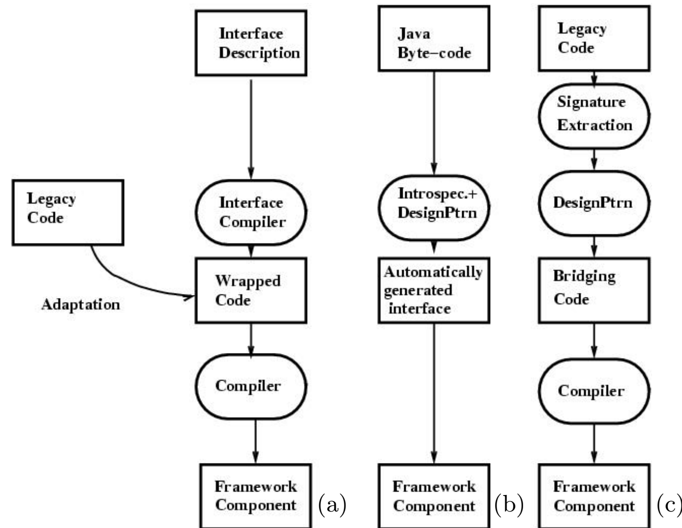

Figure6. Different approaches to component development: (a) is based on an IDL compiler; (b) is the one followed by Java Beans; (c) is the one we propose, based on automatic manipulation of legacy code.

The use of an interface description language (see Figure 6(a)), such as Babel SIDL [8] and CCM description language [9], can be very costly, in term of learning curve, code rewriting, and possibility of errors: in our opinion, present day interface description languages are taking an innatural approach, because they ask the developer to start writing an interface, and then to adapt the existing code to the automatically generated skeleton/stub files. This can be particularly time-consuming with legacy code.

As shown in Figure 6(b), JavaBeans proposes a different approach, based on introspection. Compiled JavaBeans classes are able to respond to specific queries about the methods they provide, presenting to the framework a full description of the available methods, and their interface (name, arguments and their type, returned value). Moreover, the Java Beans API sets a standard for more advanced tools, such as an explicit *Customizer* — presented to the designer when s/he wants to customize the Bean — and a redefined *BeanInfo* method, that overrides the automatically collected information.

We strongly believe design standards will emerge naturally with the advent of a Grid economy, in a way similar to what happened with the Web. Any Web-designer is free to design his/her pages in the preferred way, and any Web-browser can open them. Nonetheless, if s/he wants the web site to be discovered easily by a search engine, s/he will adhere to some HTML/XML standards, such as *META* tags, textual *ALT* tags for images and so on. Moreover, s/he will use some self-explaining description for the information provided: a car-company, for example, should add, somewhere in the web site, in a clear textual form, the

words “car” and “company”. Similarly, a component developer will choose to adhere to some well known interface for the service s/he wants to publish.

We propose a new approach, such as the one in Figure 6(c): legacy code is analyze in order to extract the signature of functions and variables; then, design conventions are recognized in it, and are used to develop the bridging code, needed to link the software unit with the framework and the other components at hand. We explore this opportunity below.

Extracting Information from Compiled Code. A direction we are investigating is the possibility of extracting information about methods signature from the compiled code, in a way similar to Java Beans. There are very effective tools for code developed with some modern languages such a C#, Lisp, Java, Visual Basic. For Java, for instance, Mocha [21] can extract full information from the code, because Java byte-code is very rich of information.

Binary code offers a more complex challenge. To reach higher performance, debugging information and many data related to the symbol table are stripped out of the executable file. On the other side, dynamically linked libraries are usually called by name: often function names and types are available in the binary. Also, the headers sometimes offered with libraries can be parsed to reconstruct some of the information needed to use the code. This is clearly a good situation: for old legacy code, headers could be missing.

As a matter of fact, some tools are available also for languages other than Java. *GDB* [22] can retrieve some information, especially if debugging data are kept. Other basic utilities for binary code (known as *binutils* [23]) are available on Unix systems.

The process we envision, is so:

- legacy code is developed with the tools of choice of the development team;
- compiled code and, if available, source code and headers, are analyzed to extract signatures of interest: methods' names and types are discovered in the code; to understand the basic semantics and the relationships among functions, the system looks for common naming and coding conventions (as for Java Beans design patterns); the application developer is offered the opportunity to choose which methods to publish, and which to keep hidden (if this is not clear from the code);
- this information can be used to generate, automatically, or with some computer-aided process: wrappers in other languages, so to allow inter-operability, featuring the minimal data conversion needed; WSDL descriptions, so to deploy the functions as Web Services; links to a graphical interface for composing functions.

Examples of Automatic Wrapping Several attempts have been done to perform automatic wrapping of existing code for particular frameworks, with noteworthy results, some of which are listed here.

Taylor et al. developed a way to wrap C function calls in Java components, to be used within the Triana framework [24]: JACAW [25] performs this operation

automatically. Before this, Mintchev and Getov [26] wrapped C libraries so to be used within a Java program: they were able to use a C implementation of MPI from Java. Also, Li et al. [27] wrapped high-performance MPI legacy code (written in C and Fortran) into Java/CORBA components, and were able to maintain very good performance after conversion. A different approach was followed by Stuer et al. [28]: a new pluglet for the H2O framework is able to export and publish all the other loaded pluglets as WSDL and GSDL documents.

We agree that Java is a very versatile tool for software distribution, and the cited works propose effective way to use Java as a standard for interface description and component linking. From the Java description, Web Services can be easily created. Nonetheless, to reach peak performance, in our opinion *ad-hoc* conversion filters among connected components should be automatically generated by the framework.

2.4 Components in a Grid Economy

A component-oriented design based on IDLs introduces a high programming overhead in the process of wrapping existing code to be re-used. The CCAffeine framework, taken as an example, requires the manipulation of several files to connect two components for a simple matrix multiplication. This is why we push for a more natural approach, based on automatic extraction of interfaces, and then computer-aided generation of suitable bridging layers. Java Beans offer an interesting example, as they base connection and discovery on interface introspection and design conventions. In our opinion, there is a need for tools able to analyze legacy code, extract signature information and then generate efficient bridging code. The strong results available in the field of wrapping C into Java components can be a valid starting point.

In the next section, we consider application and activity description. Particular attention needs to be paid to the choice of parameters that are of relevance for a component living in a Grid economy. Components' description should include information that will allow a developer to choose the fittest resource to his/her needs. This can include:

- their names and descriptions, in natural language;
- cost of using the components;
- expected (or guaranteed) performance;
- billing information (bank account...);
- access requirements (password, licenses...);
- level of data security and privacy;
- authority certifications;
- quality standards that are met;

along with more standard functional information such as:

- interface, i.e. a formal specification of services provided (used) and related parameters;

- actions, i.e. the modifications of the environment caused by the execution of the component;
- preconditions, i.e. expressions that must be true before the action(s) can take place;
- postconditions, i.e. expressions that must be true after the action(s) do take place.
- communication protocol;
- data format;
- needed computing resources.

An application developer should be able, using these parameters, to choose the components that reach the point in the cost/performance trade-off that better fits his/her needs. Also, s/he should be able to use only components which meets some standards in term of design quality, QoS and/or security and privacy. This information should allow the creation of a search engine as of that depicted in section 4.

3 Application Description and Workflows

A central concept for the deployment of Grid software components is a *workflow* description model to specify the coordination level of the activities caused by component execution. While *activities* are units of work to be performed by agents, computers, communication links etc., the *process description* of a workflow is a structure describing both the activities to be executed and the order of their execution (see also Marinescu [29]).

Workflows need to name and describe the *activities* to be composed, where activities are usually characterized by several parameters, including their name and description (in natural language), interface signature, pre- and post-conditions. These parameters are going to be used during a component search, as it will be shown in the next section.

Traditionally, workflows are encountered in business management, office automation, or production management. Grid workflows have several peculiar characteristics. For instance, *resource allocation* is a critical aspect of the Grid-based workflow enactment. The Grid provides an environment, where multiple classes of resources, within distinct administrative domains, are available. In addition, there is a large variability of resources in each class.

In this section we want to survey some of the workflow approaches currently followed in the Grid community, and compare these efforts with others in the (business) Web Services community.

3.1 Workflow Models for the Grid

In the scientific Grid community, the component model and the related workflow concepts often correspond to merely determining the execution order of sequences of tasks, which simply read/write raw files. Many scientific applications are built, in fact, by composing legacy software components, often written

in different languages, where the seamless interface is realized through raw permanent streams (files). The most common Grid workflow can thus be modeled as simple Directed Acyclic Graphs (DAGs) of Tasks, where the order of execution of tasks (modeled as nodes) is determined by dependencies (in turn modeled as directed arcs). Even if several projects have addressed the composition of specific sequences of Grid tasks, and several groups have developed visual user interfaces to define the linkage between components, currently there is no consensus on how workflow should be expressed.

Within this framework, the most notable example is the Directed Acyclic Graph Manager for Condor, *DAGMan* [30], i.e. the well known workload management system for compute-intensive jobs. Condor is a major effort to reach *high-throughput computing* on distributed resources. It features resource management and scheduling for jobs and data on large collections of computing elements. It also offers very effective scheduling for parameter-sweep applications. DAGMan is a Condor meta-scheduler, that submits jobs to the high-throughput Condor scheduler in an order represented by a DAG and processes the results.

Besides DAGs, *UNICORE* [31] provides more sophisticated control facilities in the workflow language. Constructs such as Do-N, Do-Repeat, If-then-else, and Hold-Job have been defined and integrated into the abstract job description and the client UNICORE GUI. In particular, Do-N forces the repetition of an activity N times, where N is fixed at submission time, Do-Repeat repeats an activity until a condition evaluated at runtime becomes true, If-Then-Else executes one of two activities depending on a condition evaluated at runtime, while Hold-Job suspends an activity until a given time/date has passed.

Grid Programmers have also the necessity to deploy high-performance applications made up of *peer-to-peer components* (P2P), collaborating through high-performance communication channels, characterized by standard interfaces. Unfortunately, we have to observe the lack of a standard model for specifying this kind of composition. A notable contribution to the definition of high- performance Grid components and their composition is the CCA (Common Component Architecture), introduced in the previous section. Within the CCA framework, the application structure is built as a free directed graph of port connection. It does not describe directly the movement of data, or the order of calls, but simply the set of all components which can be activated by any other components. The available commands are limited to instantiation of a component, and to connection of two *uses/provides* ports.

Within the *Java Beans* framework, the application structure can be more complex. Any component can be connected to another in order to react to changes in its status. A component, in particular, can react to user input (keyboard, mouse), time clicks, changes in the value of a property (regardless of the way it changed), explicit messages, fired events, as they appear in another bean. The network of connection can be as complex as desired, with mutual and multiple links among two Beans, including cycles and self-loops. This can built using the graphical interface of *BeanBuilder* or similar tool, and then compiled into a complete, autonomous Java application. Alternatively, the programmer can load

and connect the beans with a Java program. In this case, the language used to implement the components is the same used to connect them. As for CCA, the connections describe very loosely the logical flow of data or control dependencies among components.

3.2 Workflow Models in the Business Community

We can notice a strong similarity between these research efforts in the Grid community regarding component-based programming, and the current evolution of the Internet/Web scenario, which from a media for merely publishing static data is becoming a media of interacting *Web Services* [32–35]. In this case too, we can observe a large effort to standardize the way in which Web services (components) are published, deployed, searched and invoked. The final goal is to transform the Internet into a general platform for distributed computing as required for Business-to-Business (B2B) eCommerce. The existence of this standard is considered the central requirement to make this programming approach feasible.

In the commercial Web Service community, there are several research activities aimed to permit the specification/orchestration of a service workflow. For example, the WSFL (Web Service Flow Language) language [35], and its industry evolution BPEL4WS (Business Process Execution Language for Web Services) [36]. Also, there is commercial interest in services and protocols for registering, discovering and integrating Web Services, including the UDDI protocol [34].

An interesting comparison of Grid (CCA) and Web Service (WSDL/WSFL) programming is presented in [37]. In this paper, Gannon et al. discuss how the provides ports of a CCA component can be described by WSDL (Web Service Description Language), and hence can be accessed by any Web Service client that understands that port type. Unfortunately, no *uses* ports can be defined by using WSDL, but a distributed application can still be built by means of WSFL (Web Services Flow Language), thus defining the workflow of the application in terms of the provided WSDL interfaces. The XML-based WSFL script needs to be executed by a centralized *workflow engine* that must mediate each step of application with the various Web Services. In the CCAffeine approach, the only centralization point is the application controller component, aimed to instantiate and connect together a chain of other components. After the application is started, there is no more centralization. If the data traffic between the services is heavy, it is probably not the best choice to require it to go through a central WSFL flow engine. Furthermore, if the logic that describes the interaction between the components is complex and depends upon the application behavior, then putting it in the high level/general purpose workflow may not work. This is an important distinction between application dependent flow between components and service mediation at the level of workflow.

Despite all these problems, several Grid researchers think that it is necessary to adhere to the business standards coming from the web service world, trying to solve possible performance problems. For instance, an application described as

a composition of Web Services can be optimized by redirecting the output of a service directly to the input of another, without passing through a centralization point. This can be done without breaking the existing standards, but simply through framework optimizations.

4 GRIDLE: A Search Engine for a Component-based Programming Environment

User interactive environments like portals can provide high-level access to Grid services, such as job submission or data movement, or may offer a way to interact with specific application that have been modified to exploit Grid infrastructure. There are several classes of tools that fall into this category.

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 [38], and Grid applications have a more complex execution space than parallel programs. As previously discussed, the best results in Grid programming have been reached by scripting languages such as Python-G [39], and workflow languages such as DAGMAN [30]. 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 would like to investigate in our future 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 of other components. An example of workflow graph is shown in Figure 7. Even if this graph is flat, it has been obtained through composing different metacomponents, in particular a flight reservation one and a hotel reservation one. As you can note, we have not chosen a typical *scientific* Grid application, but rather a *business-oriented* one. This is because we are at the moment of convergence of the 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 section, we present some preliminary ideas on this issue.

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:

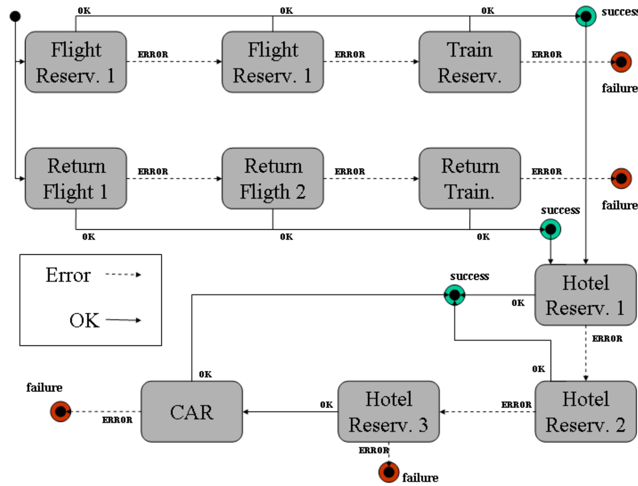

Figure7. An example of a workflow-based application for arranging a conference trip. The user must reserve two flights (outward and return) before reserving the hotel for the conference. Note that, in the case that only the third hotel has available rooms, a car is needed and must be booked too.

– 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. The place-holder component, under this model, can be thought as a *query* submitted to GRIDLE, the search engine module, in order to obtain a list of (possibly) relevant component for its specifications. 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 include also *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 discovering phase we imagine is made up of two steps. First, GRIDLE tries 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.

The last phase consists of putting together all the chosen modules in order to: (1) fill in the place-holders, and (2) *materialize* the connections among the components.

As an example, let us consider the above steps in the development process of the example depicted in Figure 7. In a Grid software development environment a programmer could have sketched the abstract workflow plan graph depicted in Figure 8.

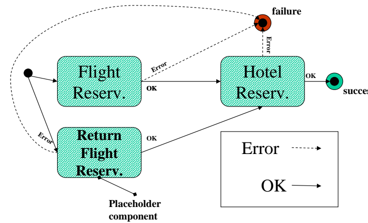

Figure8. A partially specified workflow graph, describing the application of Figure 7 at the highest level possible.

Starting from here, s/he would proceed as follow. First, s/he would look for a flight reservation component matching the place-holder. Let us suppose that such a component is available locally. GRIDLE will automatically return a pointer to it and expand the place-holder with the found component (*binding*). Figure 9 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 by 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. GRIDLE starts looking for a component. The search process is two-staged. In its first part, GRIDLE tries 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 place-holder (*binding*). Finally, when all the components are

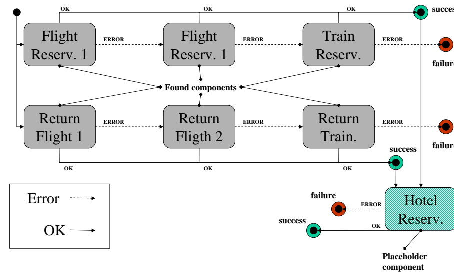

Figure9. The abstract workflow graph as it appears after the "flight reservation component" has been found.

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 components 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 here to speculate further on this opportunity. 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?

4.2 Search Techniques

In the last years, the Web Search Engine study has became 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 [40].

We would like to approach the problem of searching software components using this mature technology. We would like to exploit the concept of *ecosystem* of components to design a solution able to *discover* and *index* applications'

building blocks, and allows the search of the most relevant components for a given query.

To summarize, Figure 10 shows the overall architecture of GRIDLE,our Component Search Engine. The main modules are the *Component Crawler*, the *Indexer* and the *Query Analyzer*.

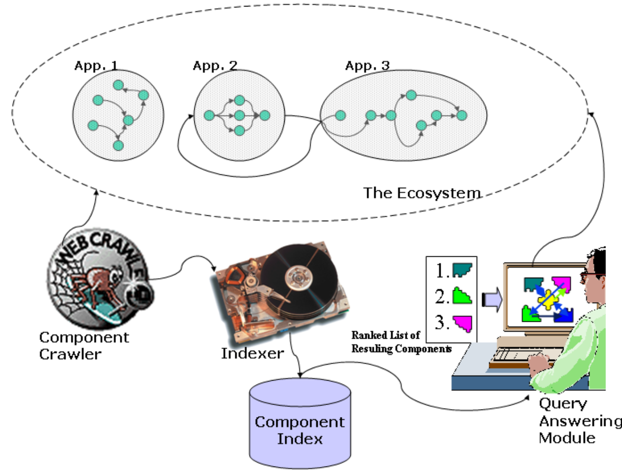

Figure10. The architecture of GRIDLE.

The *Component Crawler* module is responsible for automatically retrieve new components: (1) a pre-compiled list of repositories, (2) an incremental list of repositories continuously updated on the basis of the linking information contained within the crawled metacomponent. Consider that, in our vision, these metacomponents are workflows referring other lower-level components provided by other repositories. Note that our *ecosystem of components* lives and evolves due the these links present between 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, as interfaces and running environment requirements;
- non functional information, as 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 signature, and so on. For example, the result of the introspection operation applied to a software component. Furthermore, there are other functional features that we can notice: for example, the specification of the requirements for allowing a component to be used (e.g., *Contracts* [41]).

Non functional information consists of a textual description, in natural language, which describes the main functionalities of a component. Also, cost and performance information can be described here, along with details about guaranteed QoS, licenses that are needed for the use 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[1] 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 again the example graph of Figure 7.

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 computation of the rank according to (1) can be carried out using the well known PageRank [42] iterative algorithm. The idea of *ecosystem of components* we introduced before, is of fundamental importance for the evaluation of the components' PageRank. The main intuition behind the classical PageRank metric applied to Web Pages is that a page p should receive an importance value proportional to the probability that a random surfer will visit that page starting from another page picked at random. In our case, PageRank should be modified so that *a components will receive a higher importance value if other components are* ***actually*** *using it.* Note that in our framework, a component can be related to another either directly or indirectly (i.e. by linking to metacomponents that use that component).

After this phase, following (2) we increase the weight of "Hotel Reservation 3" by a constant factor α proportional to the relevance of "CAR".

The last module of GRIDLE is the *Query Answering* one, which actually resolves the components queries on the basis of the index. The main function carried on 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 four 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;

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

– 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.

4.3 Component Search Engines: Related Work

In the last years, thanks to technologies like Internet and the Web, a number of interesting approaches [43, 44] to the problem of searching for software components, as well as a number of interesting papers analyzing new and existing solutions [45, 46], have been proposed.

In [43], *Odyssey Search Engine* (*OSE*), a search engine for Components is presented. OSE 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 display 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.

In [44], the *Agora* components search engine is described. 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 software 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 kind of components: JavaBeans, and CORBA components. The task of searching and retrieving components is split into two distinct steps: in the first, a keyword-based search is performed and then, once the results have been presented to the user, s/he can refine or broaden the search criteria, based on the number and quality of matches. One of the most interesting features of Agora is the capability of discovering automatically the sites containing software components. The technique adopted to automatically find components is quite straight-forward 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.

In [45], 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 thirty-five subjects that rated the different used methods, 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 [47]. Among others, the most important conclusion cited in the paper is that no method did more than moderately well in terms of search effectiveness, as measured by recall and precision.

In [46], the authors cite an interesting technique for ranking components. 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 [48] to rank Web pages: PageRank [42]. In ComponentRank, in fact, the importance of a component[2] is measured on the basis of the number of references (imports, and method calls) other classes make to it.

5 Conclusions

In this contribution, we presented our vision of a new strategy to design component-based Grid applications. A three-staged approach, driven by graphical tools and accessible to non-expert programmers, should be as follows:

1. *application sketching*, based on the definition of an *application workflow*, where *place-holders* describe the needed services;
2. *components discovery*, which is performed by retrieving components matching the place-holders through a component search engine;
3. *application assembling/composition*, the phase when the component interfaces are compared and suitable filters are deployed to make types and protocols match.

The main innovation of this approach 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, based on information retrieval techniques, 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 the existence of 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.

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

When all these services become available, building a Grid application will become a straight-forward 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 as a virtual machine, tapping the power of a vast numbers of resources.

Acknowledgments

We would like to thank our colleagues at the ISTI High Performance Computing Laboratory, Ranieri Baraglia, Tiziano Fagni, Alessandro Paccosi, Antonio Panciatici, Raffaele Perego, and Nicola Tonellotto for their critical and constructive support during the preparation of this contribution.

This work was partially supported by the Italian MIUR FIRB Grid.it project (RBNE01KNFP) on High-Performance Grid Platforms and Tools, and by the MIUR CNR strategic Project L499/1997-2000 on High-Performance Platforms. Part of our research was performed in collaboration with the Computer Architecture group, led by professors Marco Vanneschi and Marco Danelutto, at the Computer Science Department, University of Pisa.

References

1. Ritchie, D.M.: The evolution of the UNIX time-sharing system. LNCS **79** (1980)
2. Szyperski, C., Pfister, C.: WCOP '96 Workshop Report, ECOOP 96 Workshop Reader. (1996)
3. Armstron, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinsk, B.: Toward a common component architecture for high-performance scientific computing. In: Conference on High Performance Distributed Computing. (1999)
4. Hamilton, G.: Java beans, api specification, version 1.01-a (1997)
5. Bernholdt, D.E., Elwasif, W.R., Kohl, J.S., Epperly, T.G.W.: A component architecture for high-performance computing. In: Workshop on Performance Optimization for High-Level Languages and Libraries. (2002)
6. Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D., Bramley, R.: Xcat 2.0: A component-based programming model for grid web services. Technical Report Technical Report-TR562, Department of Computer Science, Indiana University (2002)
7. Johnson, C., Parker, S., Weinstein, D., Heffernan, S.: Component-based problem solving environments for large-scale scientific computing. Journal on Concurrency and Computation: Practice and Experience (2002) 1337–1349
8. Kohn, S., Dahlgren, T., Epperly, T., Kumfert, G.: Babel scientific interface description language (2002) http://www.llnl.gov/ CASC/ components/ babel.html.
9. Ruiz, D.S.: Corba and corba component model (2003) http://ditec.um.es/ ~dsevilla/ ccm/.
10. Katz, D., Rasmussen, C., Kohl, J., Armstrong, R., McInnes, L.: Cca tutorial at the cca forum winter meeting. http://www.cca-forum.org/tutorials/2003-01-15/index.html (2003)

11. Sun Technologies Inc.: Java 2 enterprise edition 1.3.1 api specification (2003) http://java.sun.com/j2ee/sdk_1.3/techdocs/api/.
12. Keahey, K., Gannon, D.: Pardis: Corba-based architecture for application-level parallel distributed computation. In: Supercomputing '97. (1997)
13. Beckman, P.H., Fasel, P.K., Humphrey, W.F., Mniszewski, S.M.: Efficient coupling of parallel applications using paws. In: Proc. of 7th IEEE International Symposium on High Performance Distributed Computing. (1998)
14. Keahey, K., Beckman, P.H., Ahrens, J.: Ligature: Component architecture for high performance applications. In: International Journal of High Performance Computing Applications, Special Issue on Performance Modeling — Part 2. Volume 14. (2000)
15. Snelling, D., Tuecke, S.: The open grid service infrastructure forum (2003) http://www.gridforum.org/ogsi-wg/.
16. Microsoft Corp.: The microsoft .net framework (2004) http://www.microsoft.com/net.
17. Ximian Inc.: The mono project (2003) http://www.go-mono.com/.
18. The Java Grande Forum: http://www.javagrande.org (2003)
19. Moreira, J.E., Midkiff, S.P., Gupta, M., Artigas, P.V., Snir, M., Lawrence, R.D.: Java programming for high performance numerical computing. IBM Systems Journal **39** (2000) 21–
20. Kielmann, T., Hatcher, P., Boug, L., Bal, H.: Enabling Java for high-performance computing: Exploiting Distributed Shared Memory and Remote Method Invocation. Communications of the ACM **44** (2001) 110–117 Special issue on Java for High Performance Computing.
21. van Vlie, H.: Mocha, java decompiler (2003) http://www.brouhaha.com/ ˜eric/ computers/ mocha.html.
22. Free Software Foundation: The gnu project debugger (2003) http://www.gnu.org/ software/ gdb/.
23. Free Software Foundation: The gnu project binary utilities (2003) http://www.gnu.org/ software/ binutils/.
24. Taylor, I., Davies, R., Marzi, H.: Automatic wrapping of legacy code and the mediation of its data. In: Proceedings of the UK eScience All Hands Meeting. (2002)
25. Huang, Y., Walker, D.W.: Jacaw-a java-c automatic wrapper tool and its benchmark. In: International Parallel and Distributed Processing Symposium(IPDPS). (2002)
26. Mintchev, S., Getov, V.: Automatic binding of native scientific libraries to java (1997)
27. Li, M., Rana, O.F., Walker, D.W.: Wrapping mpi-based legacy codes as java/corba components. Future Generation Computer Systems **18** (2001) 213–223
28. Stuer, G., Sunderam, V., Broeckhove, J.: Towards ogsa compatibility in alternative metacomputing frameworks. (2004) Submitted to ICCS 2004. Available courtesy of the authors.
29. Marinescu, D.C.: A grid workflow management architecture. (2003) Proposal to the GCE and the GSM Research Groups. Available at http:// www-unix.gridforum.org/ mail_archive/ gce-wg/ 2002/ Archive/ msg00402.html.
30. Thain, D., Tannenbaum, T., Livny, M.: 11 - Condor and the Grid. In: Grid Computing: Making The Global Infrastructure a Reality. John Wiley (2003) 299–335
31. Erwin, D.: Unicore Plus Final Report. (2003) http://www.unicore.org/ forum/ documents.htm.

32. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., Winer, D.: Simple object access protocol (soap) 1.1. Technical report, W3C (2003) Available at http://www.w3.org/TR/SOAP/.
33. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (wsdl) 1.1. Technical report, W3C (2003) Available at http://www.w3.org/TR/wsdl.
34. Bryan, D., *et al.*: Universal description, discovery and integration (uddi) protocol. Technical report, W3C (2003) Available at http://www.uddi.org.
35. Leymann, F.: Web services flow language (wsfl). Technical report, IBM (2003) Available at http://www-4.ibm.com/ software/ solutions/ webservices/ pdf/WSFL.pdf.
36. Andrews, T., *et. al*: Specification: Business process execution language for web services version 1.1. Technical report, IBM (2003) Available at http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.
37. Gannon, D., Ananthakrishnan, R., Krishnan, S., Govindaraju, M., Ramakrishnan, L., Slominski, A.: 9. In: Grid Web Services and Application Factories. Volume Grid Computing: Making the Global Infrastructure a Reality. Wiley (2002)
38. Yang, C.S.D., Pollock, L.L.: All-uses testing of shared memory parallel programs. Software Testing, Verification, and Reliability Journal (2003) 3–24
39. Jackson, N.: pyglobus: a python interface to the globus toolkit. Concurrency and Computation: Practice and Experience **14** (2002) 1075–1084
40. Baeza-Yates, R.A., Ribeiro-Neto, B.A.: Modern Information Retrieval. ACM Press / Addison-Wesley (1999)
41. Holland, I.M.: Specifying reusable components using contracts. In Madsen, O.L., ed.: Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP). Volume 615., Berlin, Heidelberg, New York, Tokyo, Springer-Verlag (1992) 287–308
42. Brin, S., Page, L.: The Anatomy of a Large–Scale Hypertextual Web Search Engine. In: Proceedings of the WWW7 conference / Computer Networks. Volume 1–7. (1998) 107–117
43. Braga, R., Werner, C., Mattoso, M.: Odysseysearch: An agent system for component. In: The 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, Portland, Oregon - USA (2003)
44. Seacord, R., Hissam, S., Wallnau, K.: Agora: A search engine for software components. Technical Report ESC-TR-98-011, Carnegie Mellon - Software Engineering Institute, Pittsburgh, PA 15213-3890 (1998)
45. Frakes, W.B., Pole, T.P.: An empirical study of representation methods for -reusable software components. IEEE Transactions On Software Engineering **20** (1994) 617–630
46. Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M., Kusumoto, S.: Component rank: relative significance rank for software component search. In: Proceedings of the 25th international conference on Software engineering, Portland, Oregon, IEEE, IEEE Computer Society (2003) 14–24
47. Van Rijsbergen, C.: Information Retrieval. Butterworths (1979) Available at http://www.dcs.gla.ac.uk/Keith/Preface.html.
48. The Google Search Engine: http://www.google.com (2003)