Component-based Approach for Programming and Running Scientific Applications on Grids and Clouds

Maciej Malawski[1], Tomasz Gubała[2,3], Marian Bubak[1,3]

[1]Institute of Computer Science, AGH,

Mickiewicza 30, 30-059 Kraków, Poland

[2]ACC CYFRONET-AGH

Nawojki 11, 30-950 Kraków, Poland

[3]Informatics Institute, Universiteit van Amsterdam,

1081 HV Amsterdam, The Netherlands

Email: {malawski,bubak}@agh.edu.pl

Abstract

The paper presents an approach to programming and running scientific applications on the grid and cloud infrastructures based on two principles: the first one is to follow a component-based programming model, the second is to apply a flexible technology which allows virtualizing the underlying infrastructure. The solutions described in this paper include high-level composition and deployment consisting of a scripting-based environment and a manager system based on an architecture description language, dynamically-managed pool of component containers, interoperability with other component models such as Grid Component Model (GCM) [Baude *et al.* , 2009]. We demonstrate how the proposed methodology can be implemented by combining the unique features of the Common Component Architecture (CCA) [Armstrong *et al.* , 2006] model together with the H2O [Kurzyniec *et al.* , 2003] resource sharing platform, resulting in MOCCA component framework [Malawski *et al.* , 2005]. Applications and tests include data mining using Weka library, Monte Carlo simulation of formation of clusters of gold atoms, as well as a set of synthetic benchmarks. The conclusion is that the component approach to scientific applications can be successfully applied to both grid and cloud infrastructures.

Keywords: grid computing, cloud computing, component programming, programming model, distributed application

1 Introduction

Recently, such paradigms of scientific investigations, as e-Science and *system-level* science, have been established [Foster & Kesselman, 2006]. E-Science applications have many common properties: they are compute- and data-intensive, custom-developed by scientists using many programming languages, and used in dynamic scenarios – *experiments* – which involve various levels of coupling and composition types such as parallel or workflow processing. Grid infrastructures like EGI, DEISA, Grid'5000, Open Science Grid, TeraGrid, are now considered the key technological platforms enabling the realization of the e-science paradigm [Schwiegelshohn *et al.* , 2010]. Additionally, there is an evolution from simple computing (metacomputing) infrastructures supporting batch processing to more advanced software systems which provide high-level services. Recently, cloud computing gains attention from the point of view of scientific applications [Vecchiola *et al.* , 2009, Deelman, 2010]. Problems such as access to computation, deployment and application management still remain a challenge, due to some inherent features of the grid and cloud environments.

The main objective of research presented in this paper can be stated as follows: *How to program and run e-science applications on the grid and cloud infrastructures*? Although significant effort is being invested in research on programming models, tools and environments, the problem remains challenging [NGG Group, 2004]. The answer to this question will result in a *methodology*, consisting of a set of methods and tools, possibly integrated into a programming environment characterized by the following features: (1) facilitating high-level programming; (2) facilitating deployment on shared resources; (3) scalable to diverse environments; (4) communication adjusted to various levels of coupling; (5) adapted to the unreliable distributed environment; (6) interoperable; (7) secure.

This methodology requires an appropriate *high-level* programming and execution environment based on an appropriate programming model and supported by specific tools and services. An environment supporting these features will simplify the usage of complex computing infrastructures for people involved in e-Science. In this paper, we describe how such an environment can be built following the component-based approach and we explain why we have chosen CCA as a component standard. In order to provide a virtualization layer to the environment, we selected H2O resource sharing platform which gives us several benefits thanks to the unique features it offers.

The experience with interactive applications in CrossGrid [Bubak *et al.* , 2003], workflow applications [Bubak *et al.* , 2005, Gubala *et al.* , 2006] and virtual laboratories [Sloot *et al.* , 2006] gave us the opportunity to verify different approaches to constructing such applications. The component-based approach was investigated in various aspects: MOCCA [Malawski *et al.* , 2005] is an implementation of the CCA standard using the H2O platform which provides a lightweight container for components. The GridSpace environment [Malawski *et al.* , 2008] provides a high-level scripting approach for rapid exploratory programming and it integrates multiple technologies, including services, components and batch jobs [Malawski *et al.* , 2010]. We also conducted experiments with deployment of component applications on grid infrastructures [Malawski *et al.* , 2006b] and applicability of P2P overlay networks for providing communication in the environment [Jurczyk *et al.* , 2006]. Recent experiments with Amazon EC2 compute cloud indicate that the proposed component-based approach fits the cloud computing model as well.

The main contribution of this paper is to present the complete overview of carefully designed methods and tools which, combined together, support the component-based approach of e-Science applications development. Interconnected, they constitute a complete and self-sufficient programming and execution environment for scientific applications. We discuss the advantages and disadvantages of the component-based approach based on our experience and the lessons learned.

The paper is organized as follows: after the analysis of the state of the art (Section 2) we underline the advantages of using a component programming model and the rationale of choosing CCA and H2O as base technologies (Section 3). Next, we discuss in detail how all the requirements are met by the methods and tools we develop. In Section 4 we present case studies with model scientific applications as well as results of the benchmarks demonstrating the correctness of the approach taken. Finally, in Section 5 we give the conclusions and future work.

2 State of the art

The programming models which can be used to map computations performed by a program onto the distributed nodes of the grid come from parallel and distributed computing, and include task processing (e.g. PBS, Globus Toolkit), message passing (MPICH, OpenMPI), distributed objects (including active objects as in ProActive [Baduel *et al.* , 2006]), tuple spaces including JavaSpaces or HLA, and component- or service-oriented models. Task processing models require to use many low-level techniques such as scripting and system tools to build and run their applications. For message passing, the lack of support for application deployment in the programming model and no mechanisms for high-level composition remain drawbacks of MPI. The main drawback of distributed object systems such as CORBA is the tight coupling between objects in terms of dependencies, which becomes an obstacle for adaptability and flexibility of applications. On the other hand, a component and service-oriented models provide better support for third-party composition and reconfiguration of applications.

On a high level, a programming model defines how the whole application can be *composed* from basic blocks to provide the functionality required by the users. One deals with *composition in space* when there are many application units running (possibly in parallel) and they need to interact with one another using direct links. In component-based systems, there are several techniques of composition: low-level API as in CCA [Armstrong *et al.* , 2006], scripting languages, descriptor-based programming (ADL as in Fractal [Bruneton *et al.* , 2006]), skeletons and high-order components, and graphical tools. Popular MapReduce model [Dean & Ghemawat, 2008] also belongs to this class. *Composition in time* takes place when there are several tasks (or service operations) which have to be executed in the order of their temporal dependencies. Usually, there is a need for some external execution (workflow) engine which triggers activities and controls the order of execution. Many workflow systems are available for grids, including Kepler, Triana, Pegasus and K-WfGrid [Gubala *et al.* , 2006] systems. On the other hand, the scripting languages [Ousterhout, 1998] are useful for that purpose, since they provide constructs such as pipes and loops which allow expressing the complex control flow of the program.

In addition to composition in space and composition in time we should mention parallel and structured component composition. One approach is investigated in the CCA [Armstrong *et al.* , 2006] model, taking into account such issues as data redistribution for MxN component connections [Bertrand *et al.* , 2005]. Parallel extensions to component models are introduced to Corba Component Model (CCM) [Perez *et al.* , 2003]. Another type can be useful for more distributed and loosely-coupled scenarios, and appears as *component collections*, as in XCAT framework [Govindaraju *et al.* , 2003] or ProActive implementation of the Fractal component model [Baduel *et al.* , 2006] with Grid Component Model (GCM) [Baude *et al.* , 2009] extensions, including collective interfaces [Baude *et al.* , 2007]. The skeleton approach can be used as in ASSIST [Aldinucci *et al.* , 2005] and HOC-SA [Duennweber & Gorlatch, 2004]. The choice of the underlying programming model can restrict the high-level composition types available for applications. For instance, the component model can support all composition types, whereas e.g. pure service-oriented models do not allow (or not directly support) composition in space. The examples of XCAT [Govindaraju *et al.* , 2003] and ICENI [Mayer *et al.* , 2003] frameworks suggest that it is possible to combine both composition types in a single high-level model. Perez [Bouziane *et al.* , 2008] suggests a graph-based notation which does not necessarily imply a simple solution. It is noteworthy that composition can be also applied to Web services, as it is in the Service Component Architecture (SCA) [Barber, 2007].

The next challenging question can be stated as follows: *How access to computing resources can be obtained?*. In Globus and Unicore, virtualization is applied on the job processing level, whereas in Legion it reaches the higher software object level. In the case of service-oriented architectures, access to computation can be reduced to accessing a specific service. Due to this highest level of virtualization, Web service technologies can provide seamless access to computing resources, however they do not solve deployment problems. As an alternative, there is H2O [Kurzyniec *et al.* , 2003], a lightweight resource sharing platform. In H2O, resource providers only need to install an H2O kernel which serves as a basic container for deploying components, called pluglets, thus in H2O virtualization is applied at the container (kernel) level.

The e-science applications are often custom-developed and can evolve during their development and the lifecycle of the scientific experiment thus the process of application deployment becomes a challenge. Grid middleware, such as Globus Toolkit, provide the low-level means of application installation on the execution host by the mechanism of *staging*. Another technology enabling application deployment is virtualization [Sotomayor *et al.* , 2008]. A similar approach is offered by the *cloud computing* initiatives and the *Infrastructure-as-a-Service* model. Solutions such as the Amazon Elastic Compute Cloud (EC2) [Amazon.com, 2008] allow deploying and running virtual machine images on a configurable infrastructure. These solutions demonstrate that the need for software deployment and resource provisioning is important. The Web services model, while providing good mechanisms for accessing remote services in a loosely-coupled way, does not define any standard mechanisms for service deployment. Component technologies include the deployment process *directly into the programming model* and in the standards [Object Management Group, Inc., 2006, Baude *et al.* , 2009], since a component by definition is the basic unit of deployment. The H2O platform, on the other hand, provides a lightweight deployment mechanism. It uses Java dynamic classloading features which allow deploying and launching any Java classes published remotely (and possibly packaged as JAR files) on HTTP or FTP servers.

The job processing model, although widely supported in grids, does not offer composition in space and communication between jobs other than via inter-job dependencies. MPI model does not provide high-level composition mechanisms due to rather static application model. Distributed objects lack deployment and composition support in the model, so these important features need to be externally provided. The component model compares favourably to others, since it supports composition *and* deployment directly in the model.

3 Basic methods and tools

Having selected the component model in general, there is a need to focus on a concrete model and choose a base platform for constructing the environment. In this research the **CCA** as a component model and the **H2O** platform as a technology were selected. This decision introduces several benefits, some of which are immediate and result from the features of CCA and H2O, and some of which have to be elaborated upon and result in the ligher layers of the proposed environment.

The concept of the environment can be presented as a layered architecture, outlined in Fig. 1. On top, there is the scientific application which can be built using any of the lower layers. Below, there is a high-level composition layer, comprising two composition modes: GScript for the scripting approach [Malawski *et al.* , 2008] and the descriptor-based MOCCAccino [Malawski *et al.* , 2006a] system for composition based on the architecture description language. As discussed in Section 2, these modes are alternative approaches, so they can be used depending on the preferences of the developer and on the application type. The GScript approach is better suited for rapid development, experiments and steering, while MOCCAccino should be used for structured applications which require automated management.

The above mentioned layers are built on top of base component frameworks. MOCCA [Malawski *et al.* , 2005] is a component framework implementing the CCA model with the use of the H2O platform. MOCCA can be extended with support for the Babel system, providing programming language interoperability. Below this layer, there are basic middleware technologies which include H2O as a resource sharing platform and execution environment, infrastructure monitoring providing system status information and techniques for deployment and management of the pool of resources. The lowest layer is the grid and cloud

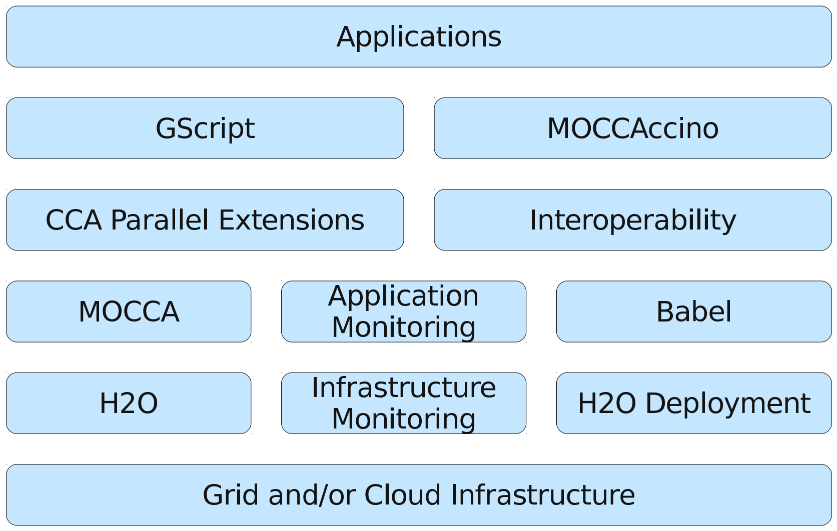


Figure 1: The layered architecture of the environment, from top: applications, high-level composition layer, parallel and interoperability extensions, base component frameworks, middleware technologies and low-level grid or cloud infrastructure.

infrastructure which may include many different middleware types, as the role of higher layers is to hide them from the component model and the application itself.

3.1 Facilitating high-level programming

The chosen programming model should allow composing the application by third parties from smaller blocks (modules) and express both temporal dependencies between them, as well as direct connections. Combination of composition in time and in space is a crucial feature of the model, since both types of interactions are present in e-science applications. Additional benefits of the component model are of a more generic software engineering nature: it facilitates code reuse, dependency management and other good practices which are often neglected in scientific programs.

In particular, CCA specifies an API for creating components and connecting their ports which can be used to provide a low-level composition in space mechanism, by using the Java API or Python and Ruby scripting. On top of it, in order to program on a high level of abstraction and to hide the details of the underlying computing infrastructure, a high-level scripting layer and an Architecture Description Language-based layer is built. The support for both models is shown in Fig. 2.

A **high-level scripting** layer is provided to enable application construction using an imperative language. By using a user-friendly API implemented in an object-oriented Ruby script, it is possible to compose the application on a high level of abstraction, while the underlying runtime system will be responsible for automatic component placement. Additionally, *the same* Ruby script (referred here as **GScript**) is used to invoke operations on the created components, using control structures (loops, conditions, iterators, etc.), hence the combined capabilities of both **composition in time and composition in space** can be expressed. The high-level scripting approach is realized as part of the **GridSpace** programming environment, where the GridSpace engine is the core of the runtime system and GRR registry stores the information of available components.

As an alternative approach, we offer an option to specify the application using a declarative language, namely an ADL. It enables hierarchical composition of component groups, where the actual number of components can be parametrized and dynamically managed. Such ADL-based composition is realized in the **MOCCAccino** system which uses HDNS [Gorissen *et al.* , 2005] registry for locating H2O kernels.

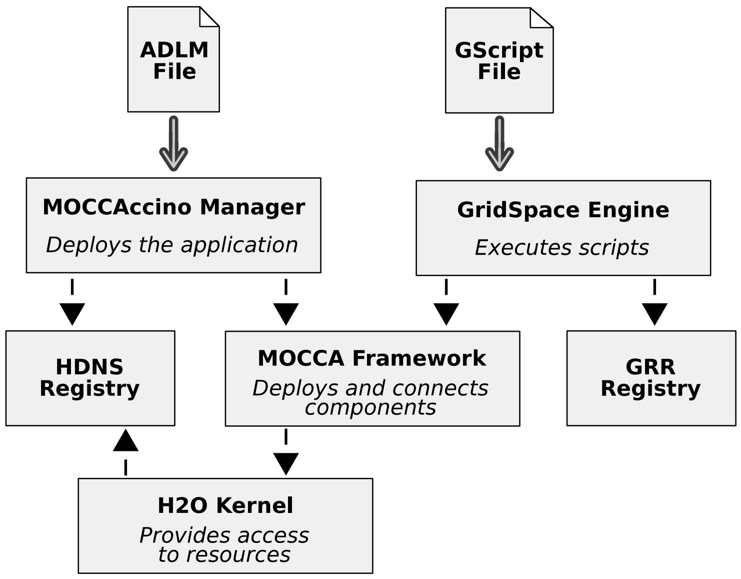


Figure 2: E-Science application composition with GridSpace and MOCCAccino as complementary scripting- and descriptor-based approaches

3.2 Deployment on shared resources

The environment should support deployment of custom application code on the available resource pool, taking into account the heterogeneity of the infrastructure and middleware. The deployment should be dynamic, allowing adaptive application behavior, namely by capabilities of deployment, undeployment and redeployment of code at runtime.

In component model, the concept of a component container and the deployment process are reflected directly. Moreover, the container provides an abstraction layer which can be used to virtualize the heterogeneous resources available, making it easier to abstract the underlying resources for the application. The selection of H2O as the component container solves the basic deployment problems, since the H2O kernel is a full-fledged application server with remote and dynamic deployment capabilities.

By selecting a component model, the problem of application deployment can be reduced to the problem of deployment of components into a container. Therefore, assuming that a pool of H2O kernels is available, the underlying grid infrastructure is *virtualized* as a pool of component containers. Using cloud terminology, the virtualization layer of H2O and MOCCA can be considered as a Platform-as-a-Service (PaaS) layer positioned above the Infrastructure-as-a-Service stack.

Unfortunately, in current production infrastructures such as EGEE/EGI, it cannot be assumed that a pool of containers is automatically available, so there is a need for a mechanism to deploy the kernels using the available grid middleware prior to actual component deployment. This approach can be seen as dynamic virtualization using a pool of transient H2O kernels created on demand and it is described in detail in [Malawski *et al.* , 2006b]. The idea is to use the concept of *pilot jobs* known from e.g. Condor [Thain *et al.* , 2005] to spawn the required number of H2O kernels as grid jobs using available middleware. Additionally, to support communication between components running in private networks of multiple clusters, it is possible to use JXTA P2P overlay network which was integrated with our system [Jurczyk *et al.* , 2006]. This solves the problem of connecting machines which cannot communicate directly because of NAT or firewall restrictions, which is often the case.

The same mechanism of dynamic provisioning of component containers is even simpler when using the cloud platform, such as Amazon EC2. We have prepared the Amazon Machine Image (AMI) with a H2O kernel installed and preconfigured to automatically start on system boot time and to listen on the public interface. Using a simple API it is thus possible to dynamically on-demand add component containers to the resource pool. It is noteworthy that adding the support for EC2 was a straightforward task, which confirms that the component-based approach fits well with the cloud infrastructure.

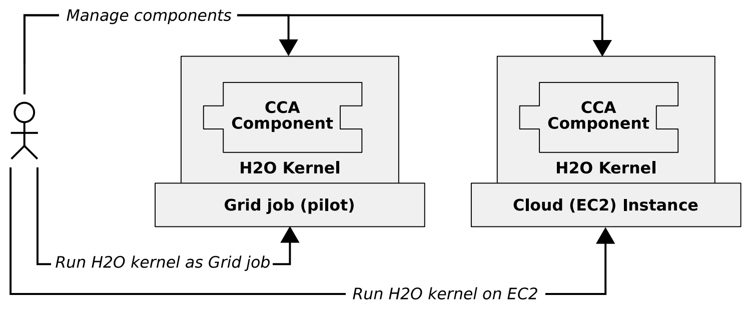


Figure 3: Deployment of component containers (H2O kernels) as pilot jobs on grid nodes or as virtual machines on the cloud. Once the pool of containers is available, the underlying infrastructure is hidden to the component framework.

The above described deployment process on grids and clouds is schematically depicted in Fig. 3. First, the user creates a pool of H2O kernels using the API for grid or cloud infrastructure. Once the kernels are running, the component application can be deployed into the kernels using standard CCA API or tools.

One observation is noteworthy with respect to the resource sharing model. Since our solution is based on H2O lightweight platform, it is possible to use resources either directly accessible via H2O, or harness additional resources from grid or cloud infrastructures by deploying H2O on top of them. This approach isolates the component application from the low-level mechanism of resource provision and for that reason it was possible to add support for new infrastructures, such as Amazon EC2 without a significant effort. This means that by creating a virtualization layer of component containers it is possible to hide the differences between grid and cloud from the application perspective.

3.3 Scalability to diverse environments

Scientific applications often involve various computation models simulated with specific, optimized environments. To support this requirement the proposed framework should be scalable to run on machines ranging from single PCs or laptops, through High Performance Computing (HPC) clusters to multiple grid and cloud sites. In other words, the environment should guarantee that the underlying infrastructure does not determine the programming model. The concept of a lightweight container and the mechanism of component composition allow creating applications in a dynamic, pluggable way, thus fitting heterogeneous environments.

In order to achieve the goal of scalability to diverse computing bases, the environment should be based on two principles: lightweight platform and mechanisms of pluggable and reconfigurable extensions. H2O can serve as a lightweight platform, since it only requires a Java 1.4 or newer virtual machine (which provides portability), runs out-of-the-box from a 20MB packaged installation, takes ca. 1 sec. to start up on a 2GHz PC and leaves a small memory footprint (approximately 25MB). This makes H2O easy to run on a developer's laptop as well as on a cluster, and easy to deploy on such infrastructures as EGI or Amazon EC2. Regarding reconfigurability, H2O provides hot deployment capabilities, while the CCA model allows for dynamic reconfiguration of component bindings at runtime. Moreover, it is possible to create new component ports at runtime, what may be useful for handling more dynamic scenarios.

3.4 Communication and levels of coupling

As the communication layer of the grid may be very heterogeneous, comprising peer-to-peer networks, WANs, LANs, inter-cluster connections, and even direct binding in a single process, the communication layer of the environment should be able to adjust the connections between the application modules to these physical constraints. The communication layer should also support collective or parallel connections between application modules.

In component model, by following the separation-of-concerns paradigm, the communication mechanism is provided by the environment, not by components themselves, thus allowing the same components to operate in both local and distributed configurations, while the protocol layer is managed by the framework. The component models also allow for parallel or group connections and communications.

H2O offers a multiprotocol communication library called RMIX for remote invocations. Therefore, it can be directly used by components in the following way: components inside a given container can use direct bindings, those located in the same LAN or cluster can use a fast binary protocol, whereas for communication over the Internet, it will be possible to switch on encryption or use the SOAP protocol wherever interoperability is required.

As applications are often parallel, there is a need to introduce some extensions to the model to support parallel connections between components. This is realized by a MultiBuilder extension [Malawski *et al.* , 2006b]. The parallel connections between component groups are handled by the MOCCAccino ADL and manager system.

3.5 Adaptability to grid and cloud environments

As e-Science applications tend to use large-scale computation components, the use of vast, powerful computing environments of the grid is a natural requirement. However, such environments may be highly dynamic and undependable, it will be crucial for the environment to provide some means of adaptability and fault tolerance. For this purpose, it should support such monitoring capabilities and adaptive features as dynamic and interactive reconfiguration of connections, locations and bindings, as well as provide support for migration and checkpointing. The component model assumes the possibility of dynamic and interactive reconfiguration of component applications, which makes it especially attractive for long-running computations within a changing environment. By restricting the application to the constraints of a component model, it is also easier to support such features as application migration and checkpointing.

There are several ways to develop a system capable of adapting to such a dynamic environment as the grid. Dynamic and interactive reconfiguration of connections, locations and bindings is directly supported by the underlying component model (CCA) and by the base platform (H2O). Some of the automatic adaptive management capabilities are reflected in the design of the MOCCAccino manager system, where it is possible to specify how a system (application) should behave when new containers are added (or removed) from the resource pool. These are handled by specific annotations in the ADL and by the adaptive behavior of the application manager. In order to be self-adaptive, a system requires some monitoring capabilities. Our concept assumes two types of monitoring: infrastructure-centric and application-centric.

3.6 Interoperability

The goal of the proposed concept is interoperability with Web services as a standard for programming distributed systems, and with the Grid Component Model which is an alternative component model supported by the CoreGRID network of excellence. By selecting H2O with RMIX which supports SOAP as one of the protocols, interoperability with Web services is in principle possible. However, the fact that RMIX does not support WSDL becomes an issue. Therefore we consider using additional Web services layer on top of H2O, so that the provided component ports can be exported as Web services using a modern embedded framework, such as XFire or Apache Axis/CXF. Since CCA is not the only one component model, and CCM and GCM are also being developed, it becomes important for the presented environment to allow components from one framework to be instantiated in a container provided by another framework and to allow inter-framework interoperability. We developed solution based on the adapter concept which enables both types of interoperability between CCA and GCM [Malawski *et al.* , 2007].

3.7 Security

Security is an important requirement for a system which allows deploying and running custom application code on remote and shared resources, including proper authentication, authorization and transport security. For large-scale systems with multiple computing nodes in multiple administrative domains, additional requirements are for Single Sign-On (SSO) and credential delegation, i.e. allowing a process running on a remote node to access resources on another one on behalf of a user.

The component model helps achieve separation of concerns by introducing the concept of a container. The security aspects such as authentication, authorization and transport security can be managed and configured by the framework. The container can provide sandboxing to protect the code running on shared computing resources from interfering with the others.

H2O kernel is a component container which provides pluggable authentication modules and flexible authorization policies. Transport security is assured by RMIX communication library which supports SSL, while the sandboxing is provided by H2O kernel using Java security features.

The first extension which we introduced into the environment was the integration of H2O with Grid Security Infrastructure (GSI) [Foster *et al.* , 1998]. This solution uses X.509 certificates with proxy extensions, which provide SSO and credential delegation, as well as compatibility with most production grid infrastructures such as EGI. As a result, the access to MOCCA framework can be granted only to such clients who can provide a valid proxy certificate of registered users [Dyrda *et al.* , 2009].

Shibboleth [Internet 2 Consortium, n.d.] is a federated Web Single Sign-On framework based on SAML (Security Assertion Markup Language). The SSO is achieved by letting users to use their home organization logins and passwords to access remote resources. Shibboleth features attribute-based access control and by mutual agreements between participating institutions it allows decentralized building of virtual organizations. We implemented the Shibboleth-based authenticator to the H2O kernel, in which a Shibboleth handle is used as a credential and an external policy decision point (PDP) is used for authorization. The advantage of Shibboleth over GSI is that the users are not required to have their certificates, but the problem is a lack of proper management of security handles for long-running computations. The Shibboleth authenticator enabled to integrate MOCCA with the virtual organization infrastructure which controls the access to the ViroLab virtual laboratory [Meizner *et al.* , 2009].

4 Case studies and experiments

In this section we present the results of the experiments which demonstrate the applicability of the component approach for the sample applications. The first experiment presents the usage of the scripting approach on the example of data mining application. The second experiment involves an application which simulates the formation of gold clusters using simulated annealing method. Finaly, we show the results of the synthetic benchmark prepared to measure the overhead of the component framework.

4.1 Weka experiments in ViroLab

The ViroLab virtual laboratory [Bubak *et al.* , 2008] is a system for collaborative construction and execution of experiments in computational science. It is focused on, but not limited to, infectious diseases caused by such viruses as HIV.

MOCCA is one of the supported middleware technologies, and the GridSpace scripting engine, as described in Section 3.1, is used as a core system for application execution. The system was applied to constructing and executing real-life examples. Below, we show how the components can be used to perform a data mining experiment using the Weka [Witten & Frank, 2005] library wrapped in components.

Key functionality elements of Weka, such as classifiers, association rules, clustering algorithms and filters, were wrapped as components to allow for more flexible creation of various experiments. To provide better performance in terms of transferring and storing datasets, it was decided to use the HTTP protocol and a WebDAV server. The components can now retrieve the datasets from any remote URL and store the results on a WebDAV server, which makes them, again, available via URL. Such *pass-by-reference* approach is very convenient, since the whole (potentially large) dataset does not have to be directly passed through the GridSpace engine.

Fig. 4 presents the scenario of an experiment which can be used to compare the performance of several classifiers from Weka on a sample dataset. It is implemented as a script shown in Fig. 5. The script demonstrates how to create an instance of a classifier component, supply it with a specific algorithm and perform the classification, measuring the time and accuracy of the predictions. The scripting approach allows easy creation of complex experiments using constructs such as loops, thus providing effective and flexible experiment *steering*.

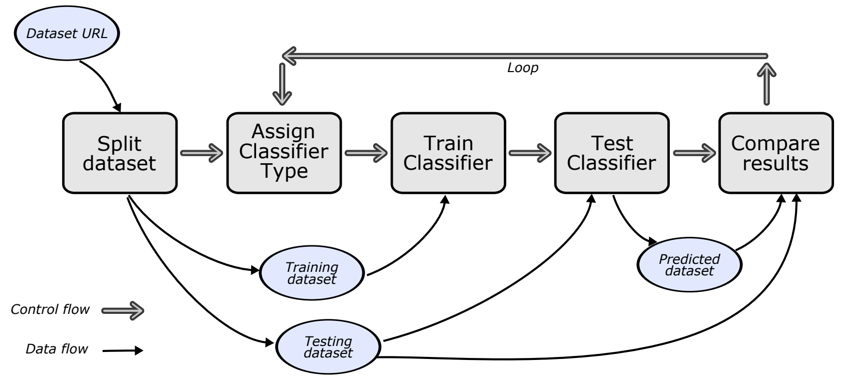


Figure 4: The data and control flow for the sample script demonstrating the use of the Weka Data Mining application which uses MOCCA components.

The experiment described above, albeit simple, demonstrates several benefits of the component-based approach. First, the *Classifier* component is a *stateful* entity, which is *created (deployed) on demand* and can use the available resources (H2O kernels). An instance of the classifier is created for each experiment run. It can also be used in *collaborative* scenarios, when a classifier is trained by one experiment (user) and then published for use by other experiments (users).

4.2 Application: Gold cluster formation

The formation of clusters of gold atoms is an important process in nanotechnology [Wilson & Johnston, 2000]. The goal is is to apply simulated annealing method to minimize the energy of the molecules, given the molecule size and the potential. The application is compute intensive, and it requires not only minimization of the energy, but it involves a larger loop, in which the actual minimization method is optimized by tuning parameters such as cooling function or initial configurations. The component-based application for simulating formation of gold clusters has evolved over time. Below, we describe a version where the energy minimization is additionally subject of an automatic tuning of the application parameters (see Fig. 6).

```
Classifiers = [
        'weka.classifiers.rules.Prism',
        'weka.classifiers.functions.Logistic',,
        'weka.classifiers.trees.J48',
        'weka.classifiers.lazy.KStar'
       ]

wekaURLgem = GObj.create(
    'cyfronet.gridspace.gem.weka.WekaURLGem')

classifier = GObj.create(
    'cyfronet.gridspace.gem.weka.WekaClassifier')

dataURL = 'primary-tumor' #address in WebDav
splitDataName = 'split-primary-tumor'
splitURLData = wekaURLgem.splitURLData(dataURL, '', splitDataName, '', 50)

i = 0
10.times do
  classifier.assignClassifier(Classifiers[i])
  learning_time = classifier.trainURLdata(
        splitURLData.trainingURLdata, '', 'class')

  classifiedData = classifier.classifyURLdata(
        splitURLData.testingURLdata, '', '', '')

  classificationPercetnage =
    wekaURLgem.compareURLData(splitURLData.testingURLdata,
                              '', classifiedData,'', 'class')

  result = classificationPercetnage.to_f * 100.to_f

  wekaURLgem.deleteURLdata(classifiedData)

  i = i + 1
end
```

Figure 5: Data mining application script. `GObj.create()` deploys the component which can be subsequently used by invoking operations directly from the script.

The *Starter* component is responsible for coordinating the work of other components. *Configuration Generator* creates the initial random configurations of atoms which are then consumed by multiple *Simulated Annealing* components, performing the actual minimization process. The *Configuration Generator* and *Simulated Annealing* components may be used for both sequential and distributed configurations, since they do not have multiple ports. The *Storeroom* component is responsible for storing all achieved configurations and may be used to derive results statistics. A single *Molecule* port is devoted to exchanging data between components. The *Storeroom* component is designed to

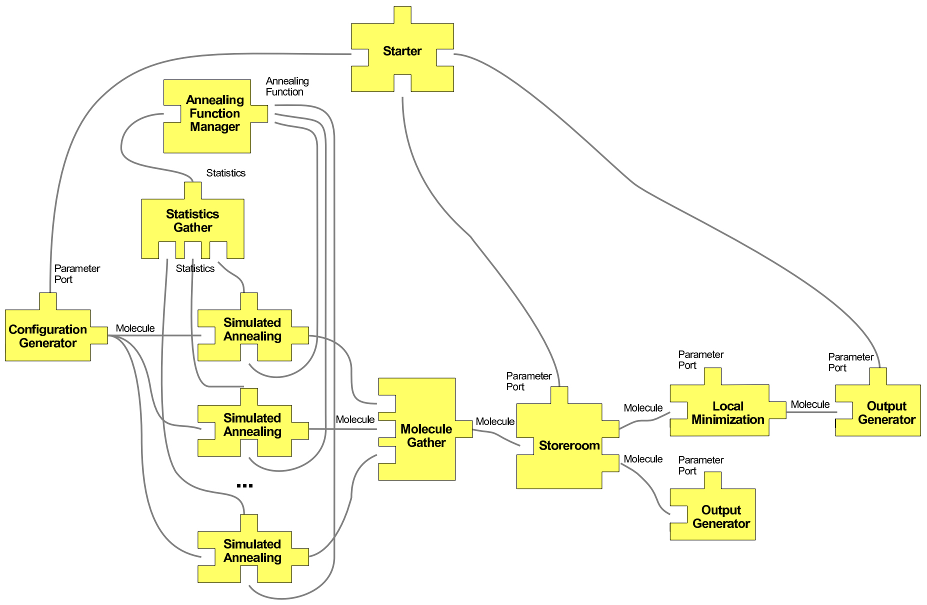


Figure 6: Configuration of gold cluster application which enables tuning its parameters in order to optimize the energy minimization process.

support a single *Molecule* provider, *Gather* component handles multiple connected components and passes their results to the *Storeroom*. This enables building a hierarchical tree of gather components, which may be required when deploying the application on a large number of nodes.

The *Simulated Annealing* components were extended to use the externally provided *Annealing Function* which represents the strategy of cooling the system and influences the optimization process. Such a function can be provided by a specialized *Annealing Function Manager* component which gathers statistics about the optimization process from the *Simulated Annealing* components in order to improve the cooling function. Additionally, the *Local Minimization* component is connected to the *Storeroom* to improve the results using the L-BFGS method (using the JAT [NASA, 2002] library). For interactive visualization, a prototype version of the *Output Generator* component was developed, using the Jmol [Herraez, 2010] visualization library (not shown in the diagram).

The *Molecule* and *Statistics* ports, together with their corresponding *Gather* components, have similar functionality. To facilitate development, a common abstract port class called *buffered port* was introduced which helps manage the queue of data items to be processed. The gather functionality has been abstracted so that it can be reused in other applications.

By following a similar approach as described in [Malawski *et al.* , 2006b], it was possible to deploy the simulated annealing application on the French Grid'5000 testbed. The application was successfully deployed on three clusters located at Sophia-Antipolis, Bordeaux and Orsay and the computing times and throughput for the molecules of 20 atoms were measured. Fig. 7 presents results of one of the experiments, showing the throughput in molecules per minute versus the number of cores used. Although the results indicate that it is possible to achieve a good speedup with our framework, the main advantage of the component approach is the flexibility of application composition and facilitated adaptation to new environments, such as Grid'5000.

4.3 Scalability experiments on Grid'5000

The purpose of the following experiments which were run on the French Grid'5000, was to test and analyze the scalability of the MOCCA environment on a large number of nodes. A benchmark application was constructed to allow extracting important system metrics, such as time of deployment, connection, invocations on collections of ports and cleanup of components.

The structure of the application is shown in Fig 8. The *Starter* component is connected to the collection of *Forwarder* components which in turn are connected to a single *Echo* component. The *echo()* operation on the port of connected components consisted of passing and returning a several-byte string

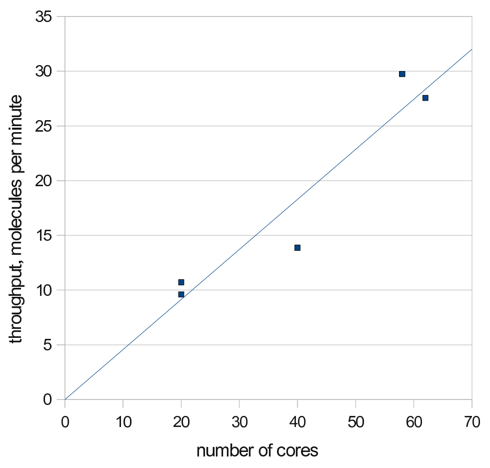


Figure 7: Throughput of the gold cluster application run on 3 clusters of Grid'5000 (Sophia, Orsay, Bordeaux)

message. The components were created using the *MultiBuilder* mechanism. The goal of ***version 1*** of the benchmark was to measure execution times where all the stages were performed *sequentially*, so no parallelism was exploited.

First, the application was run on a pool of 114 H2O kernels running on 114 nodes of 6 clusters, totalling 258 cores and the number of *Forwarder* components in the collection was equal to the number of cores. The total run time (from client startup till the end of cleanup) versus the number of cores is shown in Fig 9. It can be seen that the growth of computing time is linear with respect to the number of components (cores). This can be explained by the fact that all operations (deployment, connection, invocation and destroying) were invoked *sequentially*. The average processing time per component was 2 seconds, which is comparable to the time of running the above application on a single node. The conclusion is that creation of a large number of connections between components using the *MultiBuilder* mechanism does not introduce additional overhead. This

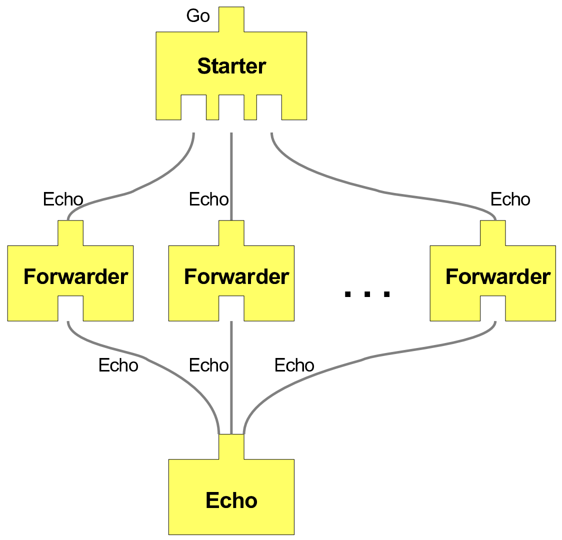


Figure 8: Configuration of components in the benchmark application. The number of *Forwarder* components in the collection is parametrized.

means that the environment preserves scalability when handling collections of components sequentially.

The goal of the second experiment was to measure the time of each stage of the benchmark application. The results of the two sample runs are shown in Tab. 4.3. As can be seen, the most time-consuming stages are the creation of components and the actual computing which is the time of passing the echo message from *Starter* through *Forwarders* to *Echo* and back again. The creation time is relatively long, since it involves opening new sessions to H2O kernels and instantiating a new component, including classloading. The reason behind the lengthy computation time stems from the implementation of CCA `connect()` and `getPort()` methods in MOCCA. When components are connected, the *uses* side only receives a reference to the *provides* side. The actual opening of a session to the H2O kernel of the provider is performed when the user component requests a reference to the *uses* port from the framework, which is done during

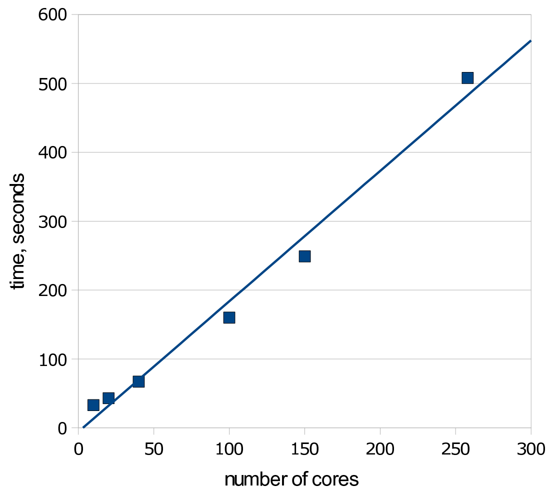


Figure 9: Total execution time of the benchmark application (version 1) on 258 cores, 6 clusters

n	creation	connection	computing	destroy	total
260	207	25	219	99	551
240	90	20	171	103	384

Table 1: Detailed measurements of application stages (version 1) for sample runs. Number of computing nodes (cores) is denoted by n and the time is given in seconds.

application execution (compute time). In the case of the benchmark application, there are two such operations per each *Forwarder*, which explains the delay and overall time.

The goal of the ***version 2*** of the benchmark was to measure the performance of parallel invocation of operations on the collection of components. The implementation of concurrent invocation in the *Starter* component is based on the *cached thread pool* executor mechanism from the `java.util.concurrent` package. The opening of sessions and execution of forwarders can then proceed in parallel. The results of detailed measurements of ***version 2*** of the benchmark performed on 100 cores distributed over 6 clusters are shown in Fig. 10.

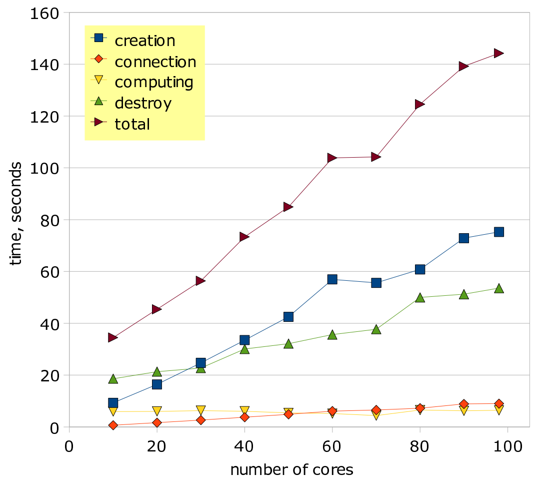
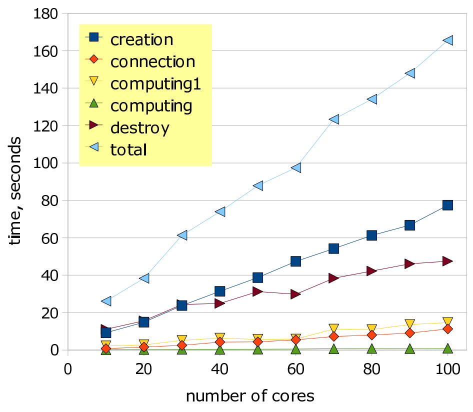


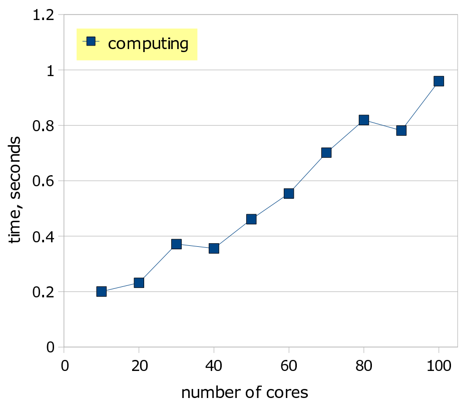
Figure 10: Detailed execution times of the benchmark application (version 2) on 100 cores of 6 clusters

This time, the computation time is reduced to approximately 5% of total run time, while for the sequential version it was nearly 50%. As expected, the asynchronous execution considerably improves the application performance, but still we can observe overhead induced by the initial opening of the connections.

In order to distinguish the opening of the H2O session from the actual remote method invocation on component ports, the *Starter* component was further modified to perform a series of invocations of the echo operation after obtaining a reference to the port (***version 3***). The time of the first invocation (labelled `computing1`) was measured separately from the average time of the 10 subsequent invocations (labelled `computing`). The results are presented in Fig. 11(a) with the computing time (enlarged scale) shown in Fig. 11(b). It can be seen that the computation time for 10 components (cores) is 0.2 seconds and for 100 it grows to nearly 1 s. The average network latency between clusters measured using the `ping` command was 0.017 s and the measured invocation time involves 4 such network hops. By comparing these values it can be seen that the component framework does not introduce significant overhead. It was observed



(a) Execution times of the stages of the benchmark application



(b) Average execution time of the computing stage (enlarged)

Figure 11: Detailed measurements of the benchmark application (version 3) on 100 cores of 4 clusters

that the invocation (computation) time grows with the number of nodes, which must be caused by the combined effects of the sequential nature of initiating asynchronous invocations, the single network connection from *Starter* and to *Echo*, as well as a single 2-CPU node these two components were deployed on. The invocation time can be potentially further optimized by using an efficient broadcast algorithm, which was, however, not the goal of this work.

In addition to the above described benchmarks it was possible to deploy and run the test application on 600 and 800 cores of 8 clusters respectively. The results shown in Tab. 2 are in agreement with the relation observed in previous tests, although more systematic experiments would be required to confirm this behavior for large-scale deployments on more than 1000 processor cores.

n	creation	connection	computing	destroy	total
800	415	80	66	287	849
600	222	49	46	202	518

Table 2: The duration of subsequent stages of application deployment on up to 800 cores of 8 clusters. The number of cores is denoted as n and the execution time is given in seconds.

The results of the large-scale deployment experiments are very promising. First, it was possible to successfully deploy, execute and clean up the benchmark application on up to 800 processor cores of 8 clusters of the Grid'5000 testbed. The times of various steps of the application lifecycle were measured and the observed behavior was explained.

Finally, we can conclude that the component-based approach does not introduce significant overhead and the environment retains scalability even for large-scale deployments which are typical for grids and clouds. These results are consistent with those yielded by tests of other Java-based frameworks such as ProActive or Satin [van Nieuwpoort *et al.* , 2006].

5 Conclusions and future work

The main conclusion is that choosing a component model and a lightweight resource sharing platform is an appropriate solution for scientific applications on the grid and cloud infrastructures. The selection of CCA and H2O as sample technologies was motivated by pragmatic reasons, since both provide tools which facilitate development and demonstration of the prototype programming environment. Nevertheless, it is important to note that both the model and the platform are general in scope and it is possible to use other technologies than CCA and H2O. This was demonstrated in by offering high-level application composition based on a scripting approach which is technology-neutral. Moreover, the interoperability experience with GCM and ProActive shows that it is possible to combine components from many models and frameworks into one application, thus hiding the details of any specific component standard.

In CCA, interactions are limited to RPC-style invocations: a component with a *uses* port can invoke methods on the connected *provides* port. This implies a synchronous request-response model. However, some component models support asynchronous interactions directly, either as an event system (as in CCM) or as asynchronous RMI (as in GCM and its implementation in ProActive). Our experience shows that the simple RPC model of interactions is not always sufficient or convenient for many classes of applications, hence work on supporting new types of component ports remains important. Nevertheless, it should be noted that this issue emerges on the level of the base component model, while all higher-level tools for component composition proposed in this paper remain valid and usable.

The concepts and methods devised in this paper are of a general nature and can thus outlive specific technologies and the implementations. Experience gained from experiments on constructing applications from components and providing higher-level tools and abstractions will be useful for both distributed computing technologies: grids and clouds. Moreover, the methods of creating virtualization layers over heterogeneous resources will gain importance as increasingly greater numbers of resource and device types become available for solving computational problems, ranging from petascale supercomputers, IaaS cloud providers, through gaming consoles such as PlayStation, to mobile devices. Specifically, adding new cloud providers to the resource pool would not change the way in which the application is constructed and deployed. This conclusion is in agreement with the one stated in [Schwiegelshohn *et al.* , 2010] that grid and cloud computing approaches can complement and benefit from each other.

A list of future research directions which were identified includes systematic development of supporting algorithms e.g. for deployment planning, higher-level programming support using semantic Web concepts, development of a more integrated environment to make it more usable, development of a formal model which would enable reasoning about the properties of the environment and the application etc, and better support for wrapping legacy applications as components to enable the environment to be practically applicable in more real-life applications.

6 Acknowledgments

The authors would like to express their gratitude to multiple collaborating colleagues from CoreGRID and ViroLab projects. Special thanks go to Daniel Harezlak, Michal Placek, Tomasz Jadczyk and Michal Dyrda, Jan Meizner and Piotr Nowakowski. This work was supported by UDA – POKL.04.01.01-00-367/08-00 grant from AGH. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, (see `https://www.grid5000.fr`). The access to the Amazon EC2 cloud was supported by AWS in Education grant.

7 Biographies

Maciej Malawski, Ph.D. in computer science and M.Sc. in computer science and in physics. Researcher and lecturer at the Institute of Computer Science AGH and at ACC Cyfronet AGH. Coauthor of over 50 international publications including journal and conference papers, and book chapters. Involved in the EU IST ViroLab project, where he was the leader responsible for the middleware task and for contacts with external users. Responsible for Virtual Laboratory developed in PL-Grid project. His scientific interests include parallel computing, grid systems, distributed service- and component-based systems, and scientific applications.

Marian Bubak, Ph.D., is an adjunct at the Institute of Computer Science AGH, a staff member at the ACC Cyfronet AGH, and the Professor of Distributed System Engineering at the Informatics Institute of the Universiteit van Amsterdam. His research interests include distributed and grid systems for scientific simulations. He co-authored about 230 papers. He lead the architecture team of the EU IST CrossGrid Project, he was the Scientific Coordinator of K-WfGrid Project and the member of the Integration Monitoring Committee of CoreGRID. He served as a program committee member, chairman and organiser of several international conferences (HPCN, Physics Computing, EuroPVM/MPI, SupEur, HiPer, ICCS, HPCC, e-Science'2006); he is co-editor of 17 proceedings of international conferences.

Tomasz Gubała M.Sc. in Computer Science, worked for the Section Computational Science at the University of Amsterdam, the Netherlands as a scientific programmer and computer science research assistant. He was involved in a major EU-funded project ViroLab as a chief designer of the virtual laboratory for infectious diseases. He is an external PhD student at the Section of Computational Science at the University of Amsterdam, he works at the ACC Cyfronet in Krakow as a scientific programmer and also applies his research as a part-time commercial solutions developer. His main scientific interests are semantic modelling of application domains, semantic integration of tools, distributed computing and services for eScience.

References

[Aldinucci *et al.* , 2005] Aldinucci, M., *et al.* . 2005. Components for High Performance Grid Programming in Grid.IT. *Pages 19–38 of:* Getov, V., & Kielmann, T. (eds), *Proc. of the Workshop on Component Models and Systems for Grid Applications, ICS '04*. CoreGRID. Springer.

[Amazon.com, 2008] Amazon.com. 2008. *Elastic Compute Cloud (EC2)*. `aws.amazon.com/ec2`.

[Armstrong *et al.* , 2006] Armstrong, Rob, *et al.* . 2006. The CCA component model for high-performance scientific computing. *Concurrency and Computation : Practice and Experience*, **18**(2), 215–229.

[Baduel *et al.* , 2006] Baduel, Laurent, *et al.* . 2006. Programming, Deploying, Composing, for the Grid. *In:* Cunha, José C., & Rana, Omer F. (eds), *Grid Computing: Software Environments and Tools*. Springer.

[Barber, 2007] Barber, Graham. 2007. *Service Component Architecture Home*. `http://osoa.org/display/Main/Service+Component+Architecture+Home`.

[Baude *et al.* , 2007] Baude, Françoise, Caromel, Denis, Henrio, Ludovic, & Morel, Matthieu. 2007. Collective Interfaces for Distributed Components. *Pages 599–610 of: Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), 14-17 May 2007, Rio de Janeiro, Brazil*. IEEE Computer Society.

[Baude *et al.* , 2009] Baude, Françoise, Caromel, Denis, Dalmasso, Cédric, Danelutto, Marco, Getov, Vladimir, Henrio, Ludovic, & Pérez, Christian. 2009. GCM: a grid extension to Fractal for autonomous distributed components. *Annales des Télécommunications*, **64**(1-2), 5–24.

[Bertrand *et al.* , 2005] Bertrand, Felipe, Bramley, Randall, Sussman, Alan, Bernholdt, David E., Kohl, James Arthur, Larson, Jay W., & Damevski, Kostadin. 2005. Data Redistribution and Remote Method Invocation in Parallel Component Architectures. *In: 19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CA, USA*. IEEE Computer Society.

[Bouziane *et al.* , 2008] Bouziane, Hinde-Lilia, Pérez, Christian, & Priol, Thierry. 2008. A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures. *Pages 698–708 of:* Luque, Emilio, Margalef, Tomàs, & Benitez, Domingo (eds), *Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008, Proceedings.* Lecture Notes in Computer Science, vol. 5168. Springer.

[Bruneton *et al.* , 2006] Bruneton, Eric, *et al.* . 2006. The FRACTAL component model and its support in Java. *Softw., Pract. Exper.*, **36**(11-12), 1257–1284.

[Bubak *et al.* , 2003] Bubak, M., Malawski, M., & Zajac, K. 2003. Architecture of the Grid for Interactive Applications. *Pages 207–213 of:* Sloot, P. M. A., & et al. (eds), *ICCS 2003.* LNCS, no. 2657. Springer.

[Bubak *et al.* , 2005] Bubak, M., Gubala, T., Kapalka, M., Malawski, M., & Rycerz, K. 2005. Workflow Composer and Service Registry for Grid Applications. *FGCS*, **21**(1), 79–86.

[Bubak *et al.* , 2008] Bubak, Marian, *et al.* . 2008. Virtual Laboratory for Development and Execution of Biomedical Collaborative Applications. *Pages 373–378 of: Proceedings of the 21st IEEE CBMS, June 17-19, 2008, Jyväskylä, Finland.* IEEE Computer Society.

[Dean & Ghemawat, 2008] Dean, Jeffrey, & Ghemawat, Sanjay. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM*, **51**(1), 107–113.

[Deelman, 2010] Deelman, Ewa. 2010. Grids and Clouds: Making Workflow Applications Work in Heterogeneous Distributed Environments. *International Journal of High Performance Computing Applications*, **24**(3), 284–298.

[Duennweber & Gorlatch, 2004] Duennweber, Jan, & Gorlatch, Sergei. 2004. HOC-SA: A Grid Service Architecture for Higher-Order Components. *Pages 288–294 of: Services Computing, 2004 IEEE Int. Conf. on (SCC'04)*. Shanghai, China: IEEE.

[Dyrda *et al.* , 2009] Dyrda, Michal, Malawski, Maciej, Bubak, Marian, & Naqvi, Syed. 2009. Providing security for MOCCA component environment. *Pages 1–7 of: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE.

[Foster & Kesselman, 2006] Foster, Ian, & Kesselman, Karl. 2006. Scaling System-Level Science: Scientific Exploration and IT Implications. *Computer*, **39**(11), 31–39.

[Foster *et al.* , 1998] Foster, Ian T., Kesselman, Carl, Tsudik, Gene, & Tuecke, Steven. 1998. A Security Architecture for Computational Grids. *Pages 83–92 of: ACM Conference on Computer and Communications Security*.

[Gorissen *et al.* , 2005] Gorissen, Dirk, Wendykier, Piotr, Kurzyniec, Dawid, & Sunderam, Vaidy. 2005 (Nov.). Integrating grid information services using JNDI. *In: 6th IEEE/ACM International Workshop on Grid Computing (Grid 2005)*.

[Govindaraju *et al.* , 2003] Govindaraju, Madhusudhan, *et al.* . 2003. Merging the CCA Component Model with the OGSI Framework. *Page 182 of: CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society.

[Gubala *et al.* , 2006] Gubala, Tomasz, Herezlak, Daniel, Bubak, Marian, & Malawski, Maciej. 2006. Semantic Composition of Scientific Workflows Based on the Petri Nets Formalism. *Page 12 of: e-Science'06 Proc.* Amsterdam, The Netherlands: IEEE Computer Society.

[Herraez, 2010] Herraez, Angel. 2010. *Jmol: an open-source Java viewer for chemical structures in 3D*. `http://www.jmol.org/`.

[Internet 2 Consortium, n.d.] Internet 2 Consortium. *Shibboleth system*. `http://shibboleth.internet2.edu/`.

[Jurczyk *et al.* , 2006] Jurczyk, Pawel, *et al.* . 2006. Enabling Remote Method Invocations in Peer-to-Peer Environments: RMIX over JXTA. *Pages 667–674 of:* Wyrzykowski, Roman, Dongarra, Jack, Meyer, Norbert, & Wasniewski, Jerzy (eds), *PPAM 2005*. LNCS, vol. 3911. Springer.

[Kurzyniec *et al.* , 2003] Kurzyniec, D., *et al.* . 2003. Towards Self-Organizing Distributed Computing Frameworks: The H2O Approach. *Parallel Processing Lett.*, **13**(2), 273–290.

[Malawski *et al.* , 2005] Malawski, Maciej, Kurzyniec, Dawid, & Sunderam, Vaidy. 2005. MOCCA – Towards a Distributed CCA Framework for Metacomputing. *In: IPDPS 2005*. IEEE Computer Society.

[Malawski *et al.* , 2006a] Malawski, Maciej, Bartynski, Tomasz, Ciepiela, Eryk, Kocot, Joanna, Pelczar, Przemyslaw, & Bubak, Marian. 2006a (December). An ADL-based Support for CCA Components on the Grid. *In: CoreGRID Workshop on Grid Systems, Tools and Environments*.

[Malawski *et al.* , 2006b] Malawski, Maciej, Bubak, Marian, Placek, Michal, Kurzyniec, Dawid, & Sunderam, Vaidy. 2006b. Experiments with distributed component computing across Grid boundaries. *In: Proceedings of the HPC-GECO/CompFrame workshop in conjunction with HPDC 2006*.

[Malawski *et al.* , 2007] Malawski, Maciej, Bubak, Marian, Baude, Francoise, Caromel, Denis, Henrio, Ludovic, & Morel, Matthieu. 2007. Interoperability of grid component models: GCM and CCA case study. *Pages 95–106 of: CoreGRID Symposium*. CoreGRID series. Springer.

[Malawski *et al.* , 2008] Malawski, Maciej, Gubala, Tomasz, Kasztelnik, Marek, Bartynski, Tomasz, Bubak, Marian, Baude, Francoise, & Henrio, Ludovic. 2008. High-level Scripting Approach for Building Component-based Applications on the Grid. *Pages 307–320 of:* Danelutto, Marco, Fragopoulou, Paraskevi, & Getov, Vladimir (eds), *Making Grids Work: CoreGRID Workshop*. Heraklion, Crete: Springer.

[Malawski *et al.* , 2010] Malawski, Maciej, Bartyński, Tomasz, & Bubak, Marian. 2010. Invocation of operations from script-based Grid applications. *Future Gener. Comput. Syst.*, **26**(1), 138–146.

[Mayer *et al.* , 2003] Mayer, A., *et al.* . 2003 (Sept.). ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. *Pages 627–634 of: UK e-Science All Hands Meeting*. ISBN 1-904425-11-9.

[Meizner *et al.* , 2009] Meizner, Jan, Malawski, Maciej, Ciepiela, Eryk, Kasztelnik, Marek, Harezlak, Daniel, Nowakowski, Piotr, Król, Dariusz, Gubala, Tomasz, Funika, Wlodzimierz, Bubak, Marian, Mikolajczyk, Tomasz, Plaszczak, Pawel, Wilk, Krzysztof, & Assel, Matthias. 2009. ViroLab Security and Virtual Organization Infrastructure. *Pages 230–245 of:* Dou, Yong, Gruber, Ralf, & Joller, Josef M. (eds), *Advanced Parallel Processing Technologies, 8th International Symposium, APPT 2009, Rapperswil, Switzerland, August 24-25, 2009, Proceedings*. Lecture Notes in Computer Science, vol. 5737. Springer.

[NASA, 2002] NASA. 2002. *Java Astrodynamics Toolkit (JAT)*. `http://opensource.gsfc.nasa.gov/projects/JAT/JAT.php`.

[NGG Group, 2004] NGG Group. 2004 (July). *Next Generation Grids 2 Requirements and Options for European Grids Research 2005-2010 and Beyond*. Tech. rept.

[Object Management Group, Inc., 2006] Object Management Group, Inc. 2006 (Apr). *Deployment and Configuration of Component-based Distributed Applications Specification, Version 4.0*. `http://www.omg.org/docs/formal/06-04-02.pdf`.

[Ousterhout, 1998] Ousterhout, John K. 1998. Scripting: Higher-Level Programming for the 21st Century. *Computer*, **31**(3), 23–30.

[Perez *et al.* , 2003] Perez, Christian, Priol, Thierry, & Ribes, Andre. 2003. A Parallel CORBA Component Model for Numerical Code Coupling. *The International Journal of High Performance Computing Applications (IJHPCA)*, **17**(4), 417–429. Special issue Best Applications Papers from the 3rd Intl. Workshop on Grid Computing.

[Schwiegelshohn *et al.* , 2010] Schwiegelshohn, Uwe, Badia, Rosa M., Bubak, Marian, Danelutto, Marco, Dustdar, Schahram, Gagliardi, Fabrizio, Geiger, Alfred, Hluchy, Ladislav, Kranzlmller, Dieter, Laure, Erwin, Priol, Thierry, Reinefeld, Alexander, Resch, Michael, Reuter, Andreas, Rienhoff, Otto, Rter, Thomas, Sloot, Peter, Talia, Domenico, Ullmann, Klaus, Yahyapour, Ramin, & von Voigt, Gabriele. 2010. Perspectives on grid computing. *Future Generation Computer Systems*, **26**(8), 1104 – 1115.

[Sloot *et al.* , 2006] Sloot, Peter M.A., Tirado-Ramos, Alfredo, Altintas, Ilkay, Bubak, Marian, & Boucher, Charles. 2006. From Molecule to Man: Decision Support in Individualized E-Health. *Computer*, **39**(11), 40–46.

[Sotomayor *et al.* , 2008] Sotomayor, Borja, Keahey, Kate, & Foster, Ian T. 2008. Combining batch execution and leasing using virtual machines.

Pages 87–96 of: Parashar, Manish, Schwan, Karsten, Weissman, Jon B., & Laforenza, Domenico (eds), *HPDC*. ACM.

[Thain *et al.* , 2005] Thain, Douglas, Tannenbaum, Todd, & Livny, Miron. 2005. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, **17**(2-4), 323–356.

[van Nieuwpoort *et al.* , 2006] van Nieuwpoort, Rob V., *et al.* . 2006. Adaptive Load Balancing for Divide-and-Conquer Grid Applications. *Journal of Supercomputing*.

[Vecchiola *et al.* , 2009] Vecchiola, Christian, Pandey, Suraj, & Buyya, Rajkumar. 2009. High-Performance Cloud Computing: A View of Scientific Applications. Oct.

[Wilson & Johnston, 2000] Wilson, N.T., & Johnston, R.L. 2000. Modelling gold clusters with an empirical many-body potential. *Eur. Phys. J. D*, **12**, 161–169.

[Witten & Frank, 2005] Witten, Ian H., & Frank, Eibe. 2005. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems).* Morgan Kaufmann.