COMODI: GUIDELINES FOR A COMPONENT-BASED FRAMEWORK FOR SCIENTIFIC COMPUTING

ZSOLT I. LÁZÁR, BAZIL PÂRV, ANDREEA FANEA, JOUKE R. HERINGA, AND SIMON W. DE LEEUW

ABSTRACT. Present work discusses the aspects pertaining to the change of scientific software development practices towards the paradigm of component-based programming [1]. It summarizes the symptoms that indicate the necessity of a renewal in computational sciences. The main ingredients for the solution are identified and a vision on how effective code sharing can affect future scientific research is presented. Starting from the premises of today's scientific software development a set of requirements for the framework, component descriptor language, component wiring and component repository are formulated. We claim that the community rather needs a useful tool even if of restricted use than an ultimate high-tech solution that will remain unaccessible to a community not willing to change overnight those programming practices it has been accustomed to for decades.

To the editor:

Abbreviated title: Framework for Scientific Computing

1. INTRODUCTION

1.1. **The problem.** Today, most scientists can program and do it as an everyday activity. A rich variety of hardware architectures, operating systems, software libraries, protocols, standards, languages, etc. are in place. The scientist, as computer user, trying to communicate and share "business logic" with fellow scientists

ZSOLT I. LÁZÁR, BAZIL PÂRV, ANDREEA FANEA, JOUKE R. HERINGA, AND SIMON W. DE LEEUW

has to fight the ubiquitous incompatibilities at a day-to-day basis. A large segment of the community writes its own software tools. Consequently, several people and research groups around the globe code for the same problem without knowing about each other. With the available source code the task is still major due to the fact that most programs are written for one specific problem and with no reusability considerations in mind. Scientific programs are often not properly structured and even if so, ad hoc binding standards are followed. Even if they used the same programming language, which is not really typical, adapting third party modules for personal use would prove to be a lengthy endeavor.

Many theoretical publications report on results of computer simulations. Unless, some widely popular software is used, reproducing the simulation by interested parties is more effort than most scientists would take.

This situation is referred to by Douglas Post as an actual crisis in computational sciences [2]. His report analyzes the symptoms, causes and possible solutions to this crisis. One of his conclusions is that high-performance computing should move towards newer technologies that provide better efficiency in terms of development efforts even on the cost of loosing some performance.

1.2. **The solution.** Component-based programming is one of todays' hottest topics in computer science [3]. Many view it as the holy grail of software reuse. Sharing code would provide computational sciences with at least three major benefits:

(1) *Efficiency* due to enhanced reusability, diversity and availability
(2) *Quality* due to the strong focus of expert groups, ranking systems
(3) *Reproducibility* of computer experiments by the author, referees and third party scientists

The means for achieving these goals are as follows:

(1) development environment for high-level visual programming
(2) standardized scientific component descriptor language (CDL)
(3) component developer tools facilitating the "componentization" of existing and future scientific software
(4) interactive project management and monitoring
(5) distributed component repository

Basically, a framework is needed which allows the assembling of scientific modules into a computational project depending on the services they offer. These services are defined via their interfaces documented in an XML-based Component Descriptor Language. These components are stored in a global component repository. Additional developer tools should facilitate adapting newly developed and existing code to the framework.

1.3. **The benefits.** Reuse oriented research would truly revolutionize computational sciences. In the component oriented future it is expected that...

- scientists program from scratch only when developing new algorithms and domain specific models
- reuse and integrate seamlessly other's components to support their own research
- find other's computational works in an instant and select from a large repository
- share their code with others
- reproduce results of computational studies published anywhere
- double check their own results using the same or similar algorithms developed by other authors
- have better control of complex computational projects
- component-technology will fully integrate with GRID technologies
- a top quality component layer will sediment in a few years by "natural selection"

2. State of the art

The first concrete steps towards component-based scientific computing have been made before the turn of the century. [4] describes a "standard for interoperability among high-performance scientific components". They touch upon most fundamental concepts of component-based programming in the context of high-performance computing and suggest a standard that they term as "Common Component Architecture" (CCA). Their recommendation for an interface definition language (IDL) closely follows the CORBA principles. The ideas therein are further developed in [5]. This group, however, favors an XML-based language for describing component interfaces. In both cases Java is chosen as the implementation language for the integration framework. Most of later works focus on the integration of the component-based approach into the realm of distributed computing in general and grid computing in particular. Notable efforts have been made by several member institutes of the Common Component Architecture Forum [6] for delivering different implementations of a CCA framework [7, 8, 9]. Most of the activity is centered around the Babel language interoperability tool [10] of the Lawrence Livermoore National laboratory [11]. Babel uses the Scientific Interface Definition Language (SIDL) for defining the interfaces of components implemented in any of the programming languages encountered in scientific computing. The palette includes C, Fortran and variants but also higher-level languages such as Java and Python.

Alexandria [12] is meant to be the future repository for CCA compliant components. As per now it contains no components.

Efforts of more restricted scope can be encountered in different fields of science such as life sciences [13], chemistry [14], nuclear physics [15], to name a few. The tremendous need in all computational sciences for sharing code is apparent if we consider the popularity of Netlib [16]. Netlib is a collection of mathematical

software, papers, and databases with hundreds of numerical libraries available for download. There is an enormous number of over 270 million requests that have been made to the repository. And Netlib only includes mathematical software, no modeling or simulation packages built over them. Unfortunately, there is no uniform way for reusing this large variety of modules originating from different places and being the result of independent and uncorrelated efforts.

OpenDX is a powerful, full-featured software package for the visualization of scientific, engineering and analytical data [17]. Its open system design is built on familiar standard interface environments. Its sophisticated data model offers great flexibility in creating visualizations by providing hundreds of built-in specialized functions. The user can drag and drop different data filters onto the canvas and connect output to inputs for setting up the data flow diagram (Figure 2). OpenDX automatically checks the compatibility of the interfaces. The functions in OpenDX are hard-coded into the application. One can contribute with new components by applying the prescriptions that are set for the interfaces. The new component will become available after the recompilation of the application.
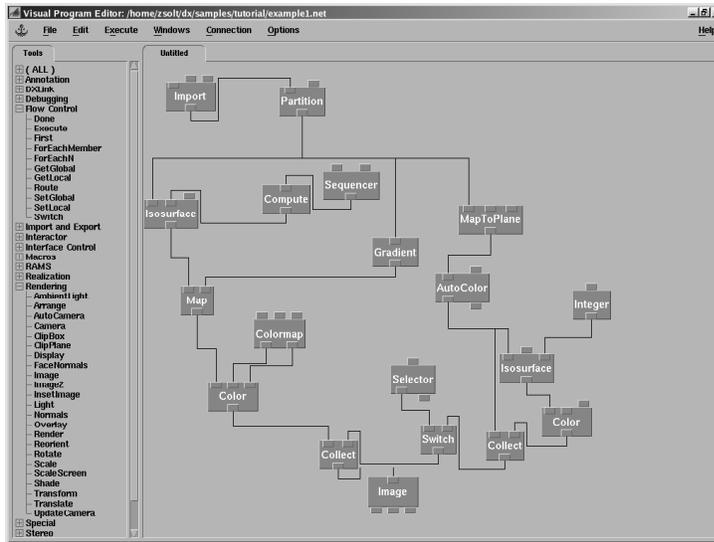

FIGURE 1. Example of flow-based programming. Representation of a data visualization project in Open Data eXplorer [17]

3. SCIENTIFIC VS. COMMERCIAL SOFTWARE DEVELOPMENT

In spite of the determining influence of natural sciences on information technology, the maturing process of computer science started with the rise of interest of

the private sector. The economical factors imposed several restrictions on using and developing software. The need for optimizing all aspects fueled studies that were beyond the goal of natural sciences. Important aspects included development and maintenance costs implying internal qualities such as reusability and external qualities such as ease of use. As the necessities of science regarding computer technologies are different from the needs of businesses and home users, their tools and programming methodologies are also different. We can even say that though natural sciences drive the evolution of IT they lag behind when it comes to using mature software development methodologies. Below, we have summarized a few of the main distinctive features of programming in natural sciences. The reader should be aware that these statements describe trends not laws. In most cases exceptions are available abundantly:

- requires scientific background of the programmer
- for a specific purpose, therefore of limited applicability
- for internal use
- does not have the constraints of commercial software development (budgets, deadlines, marketing strategies)
- performance critical
- developers use low-level languages such as C and Fortran
- high-level abstractions such as OOP concepts are rare
- requires more computer resources than commercial software (CPU, storage)
- lower complexity in terms of function points
- higher complexity algorithms
- less focus on reuse
- less investment in design
- requires less user interaction
- projects involve small to medium sized developer groups

Even though one can identify interdependencies between several items in the above list it is hopeless trying to disentangle them into cause-effect sequences. Any project that aims at changing many decades of programming traditions in computational sciences has to bear in mind these differences. Otherwise, attempts to make up the "grand unified way" will not have the proposed impact on the scientific community and will be downgraded to research of purely academic interest.

4. Requirements

Let us now formulate the expectations that we set against those constituents of a complete solution, which together will make component-based programming in computational sciences possible.

The framework should:

(1) be portable: since one of the main goals is to reduce environment dependence, a platform-independent programming is required such as Java, Python or other interpreted language.
(2) allow for high-performance computation: the implementation language of the framework should offer the facilities necessary for binding low level languages such as C and Fortran. The portability of the framework implies a higher level interpreted language. Therefore a careful design should assure a minimum involvement of the framework during project runtime.
(3) make component development and deployment easy
(4) allow for easy customization of user interfaces to domain specific needs
(5) be user friendly
(6) be open source

The component descriptor language should:

(1) describe both syntactically and semantically the component
(2) support the programming style of computational scientists as far as data structures
(3) be extensible

The component wiring mechanism should:

(1) constitute a low overhead in terms of execution time
(2) require little or no glue-code writing from the developer
(3) not require such changes in the implementation or interface of components that would make them unusable outside the framework

Certain amount of glue-code is usually inevitable. This task, however, should be the responsibility of additional developer tools.

The repository should:

(1) be distributed
(2) offer uniform and transparent access to every site storing components
(3) expose both direct user interfaces and transparent web service type access points
(4) employ a comprehensive and unambiguous classification scheme of components
(5) provide advanced search facilities
(6) provide ranking facilities of components as to popularity, user satisfaction, etc.
(7) allow for easy upload and download of components

5. Approach

Present section will discuss those guidelines that have been set forth for the COmputational MODule Integrator (COMODI) project [1]. COMODI is an international and interdisciplinary initiative attempting to offer a viable solution to computational sciences for moving towards the promised land of component technology. Henceforth, depending on the context, the word "COMODI" will refer either to the project itself, comprising the participants, ideas and means, or to the framework software with or without additional ingredients such as the component descriptor language and developer tools. Considering the faint success of present attempts it is apparent that the main challenges are not so much of technical nature but rather consist in finding the solution that is most likely to receive acceptance from the conservative community of computational scientists. COMODI's aim is to shape present programming practices such that they could metamorphose into future paradigms. In the first stage, COMODI will try to accommodate to present trends in scientific software development. This will be followed by a new generation of components that will adapt to the most recent paradigms.

COMODI's first target is the scientific software developer segment. The reasoning behind this strategy is to build up a component repository that by its sheer size will provide the variety and quality of components that will be appealing to most users. Besides, due to the level of proficiency in computer related issues, the developer segment will tolerate with more patience the less user-friendly and buggy versions in the initial phase of COMODI and will be able to contribute with important amount of know-how to its development.

Most present efforts to bring under the same roof the problematics of component-based, distributed and parallel programming are in their infancy. Indeed, this unification will most probably be ultimate solution for computational sciences. However, the surveys made by COMODI show that even though many scientists rely on parallel code the parallelism therein is usually transparent. Most don't participate in the development of parallel code and are aware of grid computing to the extent they are aware of quantum computing. Therefore we claim that effort should be invested into developing a framework, CDL and repository without mixing it with the paradigms of distributed computing.

Where does COMODI want to be better?

- short learning curve for scientific component developers
- no constructs of high-level abstraction, no new languages required, leaving that to a later stage
- bottom-up construction of the framework
- automatic glue-code generation
- zero source code line change for adapting scientific routines to COMODI

5.1. **Structure of the framework.** In order to make COMODI itself easily extensible it has to be component-based. This requires the separation of the framework into a core layer and several other modules built on the top of it. It is possible to enforce a very general view on this component architecture and deal uniformly with computational components and components that are intimately related to the framework itself. In this approach, anything apart from the core is a component, be it a simple numerical component or a heavyweight GUI. However, this uniformity, while simplifying the integration of components vital to the proper functioning of COMODI, will hit back by compromising the postulated simplicity of wiring computational components by users. Even though the customizability of COMODI to the needs of different user groups is a priority we can build upon the premise that the group of those that will contribute to COMODI will be a fraction of those that will use it or contribute to the repository. Therefore it is sensible not to sacrifice the support of user and component developer activities to those related to the development of COMODI.

COMODI will expose a core level API allowing the independent development of satellite modules such as GUIs and batch system handlers. The component architecture of the framework software itself would permit the contribution of several parties to the development of COMODI. More importantly, it will diversify the user experience allowing such "packagings" that best fit the purposes of a user group of a particular profile or closely resemble such visual development environments that this group is accustomed to, such as LabView, Simulink or OpenDX.

Under this cover COMODI, the CDL and the wiring mechanisms will be able to develop without exposing the users to major interface functionality disruptes between versions.

5.2. **Granularity.** There is no consensus as to what exactly a component and a component's port is [3]. This is due to the differences between the various languages concerning structuring code and passing data between processing units. In our case, a component can be viewed from the perspective of the computational project wherein the constituting elements are functions and procedures. If regarded from the point of view of the data that the project operates with, components should be objects in the sense of the OOP terminology. And finally, it is common to use the word "component" to a bundle of software entities, interfaces, classes, data, deployed as a unit. In other words, the physical properties of the software define the boundaries of a component.

At a lowest granularity level, where functions can be considered as components, each argument in the function signature can be considered a port. At a level of a class method, calls stand for the elementary access point while at a physical level deployment, packages can be considered the unit of assembly and whole interfaces represent the entry points. The table below is a summary of the different levels of granularity.

Component	Port
Function	argument of the signature
Class	function signature
Package	interface/pure abstract class

COMODI works at the lowest level using functions as atomic components. Even though it is meant to be a support for low-level programming, higher levels can also be emulated. Alternatively, COMODI can come with higher level APIs built over the base API. The higher level API's can be used for components that obey the rules of OOP.

5.3. **Component wiring.** In order to satisfy the requirement of low overhead in performance it is necessary that the framework does not intermediate the communication between components. Instead, it will wire up the connections by setting direct component-to-component references [10]. Therefore the components will have to comply with certain rules as to the interfaces they expose. These rules, however, should be set such that the guidelines formulated in section 4 are closely followed. [18] discusses in more detail several alternatives.

5.4. **Component Descriptor Language.** The problem of the CDL is the alpha and omega of any component-based framework. It should inspire from existing technologies such as CORBA, but at the dawn of grid computing web services are the most likely to set the standards. Given the many aspects that ought to be considered, [19], present paper will not even try to go deep into this topic. COMODI's CDL is XML-based, even though a CDL file can also have an SIDL type of representation that is more appealing to the technically trained eye [10]. The CDL's complexity is expected to grow together with the user community and the number of application areas.

6. Conclusions and outlook

The tasks to be carried out for achieving COMODI's objectives of efficient code sharing are major. It will clearly need a large user and component developer base that will bring into motion the component repository. This goal can be achieved only with a sufficiently low accommodation effort threshold. COMODI is an attempt to minimize this threshold on the cost of temporarily sacrificing generality. As such, it is not meant to be long-lived in present form. However, higher-level constructs, when the times are ripe, can be built over, hiding such obscure details that in COMODI's initial form are required for assisting a community with deep roots in "performance mining" and low-level programming. The benefits of code sharing can put scientific research on a ground that challenges science-fiction. Nevertheless, the islands of isolated feeble initiatives should coalesce into a general awareness and coherent world-wide effort. Otherwise, computational sciences are doomed to spend at least another decade in the past.

7. Acknowledgments

References

[1] *COMODI homepage,* http://phys.ubbcluj.ro/comodi/
[2] D. Post, *The Coming Crisis in Computational Science*, Proceedings of the IEEE International Conference on High Performance Computer Architecture: Workshop on Productivity and Performance in High-End Computing, Madrid, Spain, February 14, 2004
[3] C. Szyperski, D. Gruntz and S. Murer, *Component Software; Beyond Object Oriented Programming*, 2nd edition, Addison-Wesley (2002)
[4] R. Armstrong, Dennis Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker and, B. Smolinski, *Toward a Common Component Architecture for High-Performance Scientific Computing*, Proceedings of the 8th IEEE International Symposium on High-Performance Scientific Distributed Computing, August (1999)
[5] R. Bramley, K. Chiu, S. Diwan and D. Gannon, *A Component Based Services Architecture for Building Distributed Applications*, Ninth IEEE International Symposium on High Performance Distributed Computing, August 01-04, 2000 (http://www.extreme.indiana.edu/ccat/papers/hpdc2000.pdf)
[6] *Common Component Architecture (CCA) Forum homepage,* http://www.cca-forum.org
[7] *CCAFE homepage,* http://www.cca-forum.org/ baallan/ccafe
[8] *SCIRun homepage,* http://www.sci.utah.edu
[9] *XCAT homepage,* http://www.extreme.indiana.edu/xcat
[10] *Babel homepage,* http://www.llnl.gov/CASC/components/babel.html
[11] *Component Architecture for Scientific Computing homepage,* http://www.llnl.gov/CASC/components/
[12] *Alexandria component repository homepage,* http://www.llnl.gov/CASC/components/alexandria.html
[13] *IBM Life Sciences Framework homepage,* http://www-306.ibm.com/software/info/university/products/lifesciences/framework/
[14] *Octet Molecular Informatics Framework homepage,* http://octet.sourceforge.net/
[15] *ROOT homepage,* http://root.cern.ch/
[16] *Netlib homepage,* http://www.netlib.org/
[17] *Open Data Explorer homepage,* http://www.opendx.org/
[18] Zs.I. Lázár, B. Pârv, *COMODI: Component Wiring in a Framework for Scientific Computing*, Studia Babes-Bolyai, Informatica
[19] A. Mayer, S. McGough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, J. Darlington, *Meaning and Behaviour in Grid Oriented Components*, Proceedings of the Third International Workshop on Grid Computing, Springer-Verlag (2002) (www.lesc.ic.ac.uk/iceni/pdf/Grid2002.pdf)

Department of Theoretical and Computational Physics, Faculty of Physics, Babeş-Bolyai University, Str. M. Kogălniceanu Nr. 1, RO 400084 Cluj-Napoca, Romania
  E-mail address: `zlazar@phys.ubbcluj.ro`

Chair of Programming Languages and Methods, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Str. M. Kogălniceanu Nr. 1, RO 400084 Cluj-Napoca, Romania
  E-mail address: `bparv@cs.ubbcluj.ro`

Chair of Programming Languages and Methods, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Str. M. Kogălniceanu Nr. 1, RO 400084 Cluj-Napoca, Romania
  E-mail address: `afanea@cs.ubbcluj.ro`

Physical Chemistry & Molecular Thermodynamics, DelftChemTech, Technical University of Delft, Julianalaan 136, 2628 BL Delft, The Netherlands
  E-mail address: `J.R.Heringa@tnw.tudelft.nl`

Physical Chemistry & Molecular Thermodynamics, DelftChemTech, Technical University of Delft, Julianalaan 136, 2628 BL Delft, The Netherlands
  E-mail address: `S.W.deLeeuw@tnw.tudelft.nl`