COMODI: Architecture for a Component-Based Scientific Computing System

Zsolt I. Lázár[1,2], Lehel I. Kovács[1], and Zoltán Máthé[1]

[1] Babeş-Bolyai University,
400084 Cluj-Napoca, Romania
zlazar@phys.ubbcluj.ro,
[2] University College Dublin, Belfield, Dublin 4, Ireland

Abstract. The COmputational MODule Integrator (COMODI) [1] is an initiative aiming at a component based framework, component developer tool and component repository for scientific computing. We identify the main ingredients to a solution that would be sufficiently appealing to scientists and engineers to consider alternatives to their deeply rooted programming traditions. The overall structure of the complete solution is sketched with special emphasis on the Component Developer Tool standing at the basis of COMODI. Prototypes for a framework and an automatic interface description generator are presented.

1 Introduction

As signaled already in the late sixties, when the term "software crisis" got coined [2], brute computing force growing at exponential rate is not the answer to the problem of complexity. Recently, scientists find similar obstacles standing in the way of large scale scientific software projects [3, 4], and argue for a change of paradigm. These ideas can be further extended in the context of small and medium size projects that make up the bulk of the activity in the community. In [5] and [6] it is pointed out that *reuse oriented programming (ROP)* would dramatically improve the efficiency and quality of computational research. In the last decade, there has been significant progress in overcoming the thorny technical problems of component based software engineering (CBSE) in high-performance computing [7]. A number of notable efforts have been started under the umbrella of the Common Component Architecture (CCA) Forum (`http://www.cca-forum.org`) yielding a specification for the behavior and interaction of components and frameworks. Several complying solutions have been implemented ([8–10]) based on Babel, the language interoperability tool, and the associated Scientific Interface Definition Language (SIDL) [11].

While accounting for all technical questions is a necessary condition for reaching the proposed goals of CBSE this might not be sufficient for a real impact on the community's software development practices. The COmputational MODule Integrator (COMODI) is another initiative aiming at a component based framework, component developer tools and component repository for scientific

computing. The emphasis is, however, shifted toward issues pertaining to the human-computer interaction. It is based on a set of requirements formulated in the light of the conclusions of a preliminary survey made with a mixed group of computational scientists on the occasions of conferences, workshops and via an on-line form on the COMODI website [1]. Present paper briefly reviews the key concepts behind COMODI and lays out the large scale architecture of the complete suite of tools that is up to the challenges of an ROP paradigm. The formulated requirements and design strategies focus on the needs of component developers rather than those of users. In section 4 we report on the experience gathered while working with a first prototype endowed with some of the functionalities prescribed in the previous sections.

2 Requirements

For most scientists the computer is merely a tool. Therefore, whatever little time is spent on making the computer do its job this time is usually a loss from the point of view of the objectives of the project. In more and more fields, "survival" without a decent level a programming and system management skills is not possible. At a global scale the time scientists spend waiting for computing jobs to finish is a fraction of what computers spend idle waiting for the scientists to finalize the development of code. In view of these facts it is only fair to admit that "high-performance programming" should get equal attention to high-performance computing. To this end, future computing technologies should strive to make the usage of computers significantly easier. Another factor that, if ignored, can prevent a valuable software project from getting the deserved attention is the level of support for existing technologies. The new solution should be able to accommodate old-style data while newly produced data should be usable within the old framework. And finally, the new technology can only penetrate the scientific community if it is free of charge.

In order to make the above general requirements more specific we first must make distinction between *component developers* and *end-users*: the former design and implement new components while the latter are primarily involved in assembling these into executable applications.

Since the two activities require different skills and work methods the requirements set for the employed tools in each case also differ. For user satisfaction the solution has to be endowed with the features listed below:

- user friendly graphical interfaces;
- intuitive, high-level representation of data and processes such that the elements of low-level programming can also be clearly identified;
- possibility for low-level control;
- support for most popular hardware and software platforms;
- comprehensive component repository;
- high-performance;
- open source.

In order to fully support developers it is imperative that no compliance criteria are set for the computational code neither in terms of structure nor used data types. In other words, any valid code written in the supported programming languages should automatically be ready for COMODI. Therefore several restrictions apply to the process of adapting existing code to COMODI:

- no change in the source code, the interface or the implementation;
- no extra coding - connectivity is achieved by supplementing author provided source-code with automatically generated glue-code;
- no need for the author to know other languages/standards then the ones used for implementing the code;
- no platform dependence - the capabilities of the system the development is carried out on is extended by on-line servers providing compilation as web service;
- no language dependence - all present and future languages should be able to communicate seamlessly;
- low performance overhead;
- support for both open source and commercial components.

3 Architecture

The complete solution should include the following elements:

- visual programming environment for computational projects;
- component developer tools for adapting regular code to the framework;
- distributed component repository;
- compilation web service.

The above software elements will come with several standards including one for global naming of components and a language for documenting their services. Figure 1 shows the COMODI architecture. The responsibilities of each part are summarized in Table 1.

Role of the framework	**Role of the component developer tool**
– component assembling – project verification and validation – project execution – runtime user interaction	– assist the developer in documenting the code – generate glue-code – assist the developer in compiling the component – register the component with the global repository

Table 1. Responsibilities of the two major parts of COMODI

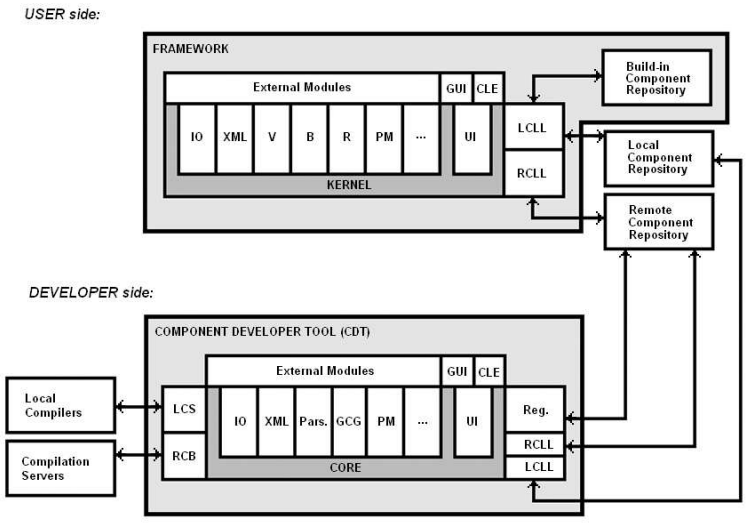

Fig. 1. Architecture of COMODI. On the user side: IO: *Input/Output System*, XML: *Extended Markup Language Parser*, V: *Validator*, B: *Binding System*, R: *Running System*, PM: *Project Manager*, UI: *User Interface*, GUI: *Graphical User Interface*, CLE: *Command Line Editor*, LCLL: *Local Component Locater and Loader*, RCLL: *Remote Component Locater and Loader*. On the developer side: LCS: *Local Compilation Service*, RCB: *Remote Compilation Broker*, Pars.: *Parser*, GCG: *Glue-Code Generator*, Reg.: *Registrar*.

The developer layer contains a user friendly *Graphical User Interface* (GUI), a *Component Developer Tool* (CDT) with a *Parser*.

The CDT, after semi-automatically collecting information pertaining to the content, behavior, and representation of the component, generates a *component descriptor file* (CDF) in the XML based *Component Descriptor Language* (CDL) and the source of the *glue-code* that will intermediate the communication of the component within the COMODI framework (see figure 2). At this stage the CDF will contain all communication related information such as exported functions and data types. It describes both syntactically and semantically the component, supports the programming style of computational scientists as far as data structures, and it is extensible. Its complexity is expected to grow together with the user community and the number of application areas. By semi-automatic we mean that the *Parser*, which stands at the basis of the tool, analyzes lexically and syntactically the source file, extracts interface information *only*, and generates a primary CDF. Using the GUI, the developer only has to confirm the exported ports, provide human readable documentation for the component, set default values and visual representation related preferences. The CDT then contacts on-line *compilation servers* and returns ready-made binaries for the platforms of the developer's choice. The compiled library together with the descriptor file

are packed into standard format, such as `tar.gz`, are uploaded by the developer to a place where it can be accessed publicly while the CDT registers the component in the *Remote Component Repository*. The deployed component is a package containing the component's source code - if the developer chooses to make the source open - the component descriptor file, the binaries for both the computational- and the generated glue-code, and further resources. Upon use within the COMODI framework, the component is downloaded and stored in the *Local Component Repository*.

The sources provided by the component developer suffer no changes during the component creation process. All glue-code comes as additional functions in a separate file. Not touching the source of the developer has the benefit of the compiled component being usable both within and outside the COMODI framework making COMODI components fully compatible with traditional programming environments.

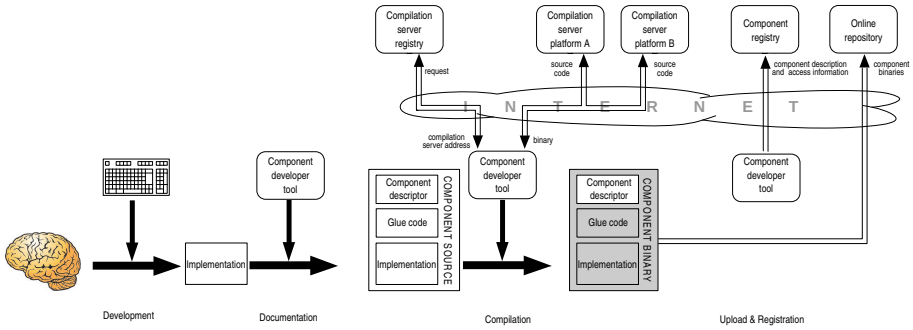

Fig. 2. Component development process

4 Prototype

4.1 Framework

Language interoperability is one of the most important challenges for CBSE in high-performance computing. Fortunately, Babel can successfully handle many of the technical difficulties. Therefore, together with the CCA standard it served as a good starting point for COMODI. Present version of the COMODI framework is a partial Java implementation of the CCA specification. Only those parts have been covered that are essential for testing the ideas of interest.

We find that the CCA specification is overly comprehensive for the average user primarily interested in elementary tasks like instantiating components, discovering their properties, linking their ports and running the so formed diagrams. Thus, a simplified add-on Java package has been created targeting the

user-framework interaction. Even though the reduced interface is mainly a subset of CCA it adds one concept to those encountered in the CCA specification, namely *diagrams*. There is an 1:N type of association relation between diagrams and components. Diagrams are independent of each other but managed by a single framework instance. The user can create components, components, connections and diagrams. Components and connections can be added and removed from diagrams. These are executed starting from a `GoPort`. In this version the only pursued goal was to keep this interface minimal and free of concepts that are not immediately obvious to the user.

Since enhanced interactivity is one of the main requirements of the ultimate solution we were testing yet another layer on the top of the reduced interface. The motivation behind the interactive layer not directly interfacing to the framework is that this layer is independent of CCA and the underlying framework both at interface and implementation level. Instead of creating a completely new interaction console like in CCAFE we chose Python for a number of reasons. To name a few: *i.* implementation and maintenance is considerably easier while the interaction is much richer. History, code completion and documentation are readily available; *ii.* the user has at hand a full-fledged and easy-to-use programming language for pre- and post-processing data and for performing auxiliary tasks such as interacting with the operating system, working with strings, etc.; *iii.* no COMODI specific syntax has to be learned by the user. A typical session would contain most of the following elements:

```
>>> from comodipython import *
>>> framework = Framework()
>>> server = framework.createcomponent("HelloWorldServer")
>>> client = framework.createcomponent("HelloWorldClient")
>>> print server.ports()
['HelloPort']
>>> print client.ports()
['GoPort', 'HelloPort']
>>> clientport = client.port('HelloPort') # uses port
>>> serverport = server.port('HelloPort') # provides port
>>> connection = framework.createconnection(clientport, serverport)
>>> diagram = framework.creatediagram('HelloWorld')
>>> diagram.add(server)
>>> diagram.add(client)
>>> diagram.add(connection)
>>> diagram.run(client.port('GoPort'))
```

4.2 Parser

The parser is implemented such that it can recognize the majority of imperative programming languages, once their EBNF definitions are available. In addition to the complete interface information available in the source file it also extracts

dependencies such as function calls made from inside other functions. This feature helps in selectively including dependences from header files. The level of flexibility with respect to the source's implementation language at the input is replicated at the output. The description of the source code's interface is generated based on a user provided table of prescribed structure. Creating SIDL, Babel XML or any other format requires almost straightforward customization of the table. Naturally, not all information for the output of a given format is available in all languages or there is no unique translation. These cases require human intervention. However, the appropriate output table can make the parser generate data that can directly be fed to the tool in charge for collecting the remaining information from the user. Alternatively, the raw XML output can be transformed into other formats and a higher-level description by a separate piece of software. A parser prototype can be found on the COMODI website.

Presently, the parser has been tested for C and Pascal while Java and Fortran 77 are expected to follow shortly as the only challenges they pose are strictly implementational. Complex hybrid-languages like C++ or Fortran 95 will most probably pose challenges [12], however, as the parser is only targeting interface information the vast majority of problems that could be encountered in case of complete parser are avoided.

5 Future work

As our primary concern is the component developer tool, most effort will go into creating a prototype that strikes a balance between the requirements of minimial human intervention during the component creation process and that of large enough scope so that it would be useful.

Presently, connecting two components requires a very tight match between the corresponding uses and provides port. Babel based frameworks cannot directly link a client component using an interface defined as `A {f:F}` to a server component providing an essentially equivalent interface `AA {ff:F}`, where `A` and `AA` are the interface names, `f` and `ff` are method names, and `F` is the type of the two methods. Even though some of the emerging issues are not merely technical, several solutions are available. With this obstacle moved out of the way one still needs more flexible subtyping rules that allow the connection of a uses port with a method such as `f(p1:T1, p2:T2)` to provides ports like `ff(p2:T2, p1:T1)` or `ff(p1:T1, p2:T2, p3:T3 = v3)`, where `p1, p2, p3` are formal parameters, `T1, T2, T3` denote data types and `v3` stands for a default value. Higher-level programming will require the framework and the component developer tools to fill in automatically or semi-automatically small incompatibility gaps like those above by generating appropriate glue-code and by hidden or user controlled connector components. We also consider that Babel's SIDL if to serve as a component IDL should be extended to cover the declaration of interfaces for uses ports.

Pushing the automation level of the interface definition via the parser will raise thorny issues such as how to disentangle environment dependences from

component interdependences and where is the borderline between the framework and the rest of the system. The ideal solution would require a consistent component based construction of the whole operating system and all third party software.

On the user side, the prototype for a GUI should ideally be preceded by a study on how the representation of component networks and their interaction with the user is preferred by the different groups in scientific computing. The interface of Ccaffeine provides a high-level view on the component diagram. Alternatively, the GUI features promoted in [6] reflect the actual low-level structure of component interfaces allowing the user to have the control that is typical for languages like C and Fortran. On the other hand, this low-level control requires extra effort from the user that on the long run might not be acceptable.

6 Conclusions

The transition of computational research toward a reuse oriented programming paradigm is conditioned not only by the existence of sophisticated tools capable of integrating several technologies but also by the extent these can bring about efficiency in the work of scientists already overwhelmed by the technicalities plaguing today's software solutions. We attribute the lack of impact of present solutions to code reuse to the high threshold of effort required, in many cases made worse by a closed source and restrictive copyrights. The paper presents the general requirements and a few design guidelines for a complete reuse oriented solution for computational scientists. We argue that the community needs a solution that allows a smooth, effortless transition to the new paradigm. The COMODI project is an attempt to identify these requirements, design complying solutions and create proof-of-concept prototypes following these principles. Our prototype framework was built on Babel and the CCA specification which proved to be powerful tools. On the other hand, we found that the tasks are not purely technical and several principle questions have to be addressed on the way toward an ubiquitous CBSE.

Acknowledgments. This work is supported by the National University Research Council of Romania with grant no. 27687/14.03.2005 and assisted by the Marie-Curie Grant no. MTKD-CT-2004-003134. One of the authors (Zs.I.L) thanks Ronan McNulty for his valuable comments.

References

1. COMODI homepage: `http://comodi.phys.ubbcluj.ro`
2. Naur, P., Randell, B.: Software Engineering: Report on 1968 NATO Conference, NATO, 1969
3. Post, D.E.: 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

4. Post, D.E., Votta, L.G.: Computational Science Demands a New Paradigm. Phys. Today, January (2005) 35
5. Lázár, Zs.I., Pârv, B., Fanea A., Heringa, J.R., de Leeuw, S.W.: COMODI: Guidelines for a Component Based Framework for Scientific Computing. Studia Babeş-Bolyai, Series Informatica, Vol. XLIX, No. 2 (2004) 91
6. Lázár, Zs.I., Fanea, A., Ciobotariu-Boer, V., Petraşcu, D., Pârv, B.: COMODI: On the Graphical User Interface. Proceedings of the 7th IEEE Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timişoara (2005)
7. Allan, B.A. *et al.*: A Component Architecture for High-Performance Scientific Computing. The International Journal of High Performance Computing Applications, Vol. 20, No. 2, (2006), 163
8. Govindaraju, M., Krishnan, S., Chiu, K., Slominski, A., Gannon, D., and Bramley, R.: Merging the CCA component model with the OGSI framework. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, (2003)
9. Allan, B., Armstrong, R., Lefantzi, S., Ray, J., Walsh, E., and Wolfe, P.: Ccaffeine - a CCA component framework for parallel computing. `http://www.cca-forum.org/ccafe/`, (2003)
10. Zhang, K., Damevski, K., Venkatachalapathy, V., and Parker, S.: SciRun2: A CCA framework for high performance computing. Proceedings of the 9th IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments, (2004)
11. Lawrence Livermore National Laboratory: Babel homepage, `http://www.llnl.gov/CASC/components/babel.html`, (2004)
12. Dan Quinlan, Qing Yi, Gary Kumfert, Thomas Epperly, Tamara Dahlgren, Markus Schordan, and Brian White: Toward the Automated Generation of Components from Existing Source Code. The Second Workshop on Productivity and Performance in High-end Computing, San Francisco, Feb, (2005)