STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLIX**, Number 2, 2004

# COMODI: COMPONENT WIRING IN A FRAMEWORK FOR SCIENTIFIC COMPUTING

ZSOLT I. LÁZÁR AND BAZIL PÂRV

Abstract. We present several alternatives for wiring atomic components, namely functions, within the COMODI framework [1]. The benefits and drawbacks of the different solutions are analyzed in the light of a few major guidelines that are set forth for COMODI. A mixture of pipe&filter and repository type of architectures is found to satisfy best the requirements for scientific computing. The role of connector components are also discussed.

## 1. Introduction

In [2], a few guidelines have been established for the possible wiring mechanisms. According to this paper the wiring should

- constitute a low overhead in terms of execution time
- require little or no glue-code writing from the developer
- not require such changes in the implementation or interface of components that makes them unusable outside the context of the framework

Even though the above requirements sound restrictive the forthcoming sections will show that it is possible to design such architectures. Let us first define the problem.

## 2. Calls & connectors

As described in [2] and ch. 8 of [4], components can be considered at different levels of granularity. For the points to be made in this paper, we shall distinguish

Received by the editors: September 2, 2004.

2000 *Mathematics Subject Classification.* 68U99, 68N99.

1998 *CR Categories and Descriptors.* D.2.12 [**Software Engineering**]: Interoperability – *Interface definition languages*; D.2.11 [**Software Engineering**]: Software Architectures – *Languages, Patterns*; D.2.6 [**Software Engineering**]: Programming Environments – *Graphical environments, Integrated environments, Interactive environments, Programmer workbench*; G.4 [**Mathematics and Computing**]: Mathematical Software – *User interfaces*; J.2 [**Computer Applications**]: Physical Sciences and Engineering – *Aerospace, Archaeology, Astronomy, Chemistry, Earth and atmospheric sciences, Electronics, Engineering, Mathematics and statistics, Physics* .

between physical and logical components. By the former we mean units of deployment, namely native library files, Java class files, etc. In view of the low-level languages used in high-performance computing, this is almost synonymous with units of compilation. Logical components are viewed from the perspective of the computational project assembled by the user. A *project* is a graph of interconnected components, as shown in Figure 1. Following the recommendations from [2], we shall work at the lowest level of granularity, i.e. the atomic components will consist in regular functions. Within the context of this work we shall use the words "component" and "function" interchangeably. *Connection* means data flow via function interfaces from the output of one component to the input of another.

We can distinguish between *horizontal* and *vertical communication*. The latter, e.g. 1.1 - 2.1.2, is the direct representation of the *pull model* for module communication. It consists in a function calling another, which returns a result to the former. This is basically the only model for communication in languages such as C or Fortran. In terms of connection architectures, it belongs to a client-server setup. On the other hand, horizontal calls imply the piping of an output into the input ports of another component. This pipe&filter architecture, and implicitely the *push model*, is not directly supported by low level languages. It is the representation of equivalent vertical calls at a high level of abstraction.

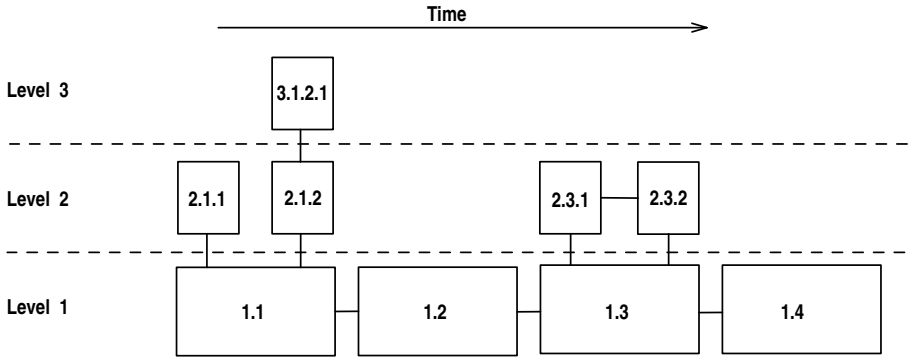


FIGURE 1. Structure of a project in a framework at a higher level of abstraction. See Figure 2 for more details.

We can define two levels of abstraction when modeling this communication. At higher level (Figure 1), we disregard the existence of the framework and certain elementary *connectors* that are components that perform simple tasks pertaining rather to the mediation of information between two or more components. This indirection can be due to reasons such as the necessity for satisfying certain syntactical requirements of component wiring. For a review on connectors see [3] and ch. 10 in [4]. At a lower level of abstraction (Figure 2), horizontal data flow is

"translated" into vertical flow by the help of connector components that we will henceforth term as *propagators*.

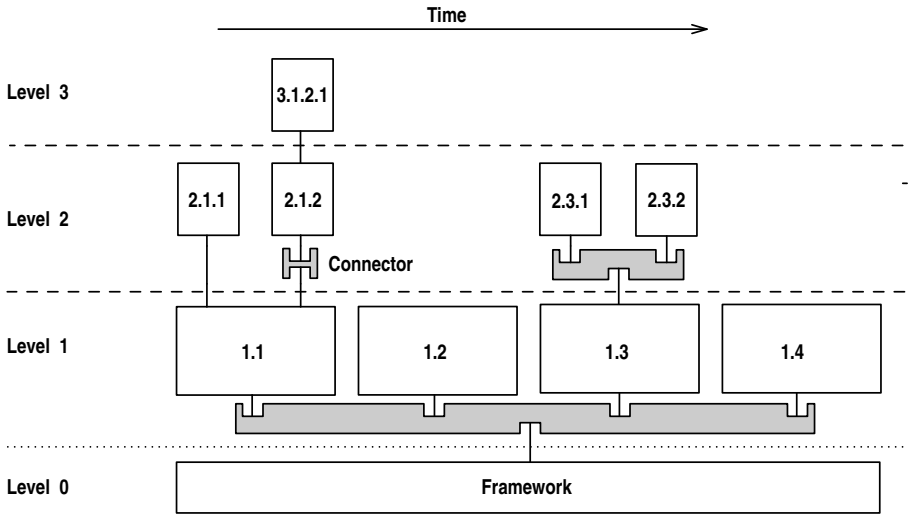


FIGURE 2. Structure of a project in a framework at a more concrete level of abstraction. The dotted line represents communication through interfaces that are discovered at runtime. The data flow between the upper layers, through the dashed line, takes place via interfaces that are established at compile time and necessarily have to be compatible. The components reach execution state starting from left to right, bottom to top. In this case the order is: framework, 1.1, 2.1.1, 2.1.2, 3.1.2.1, 1.2, 1.3, 2.3.1, 2.3.2, 1.4.

Excluding for now more advanced flow-control such as conditional branching and loops, the connection of components at low abstraction is tree-like since there are only vertical calls. At the lowest level, the framework calls the component positioned at level one. These can call others at higher level and so on. The layering is strict in the sense that there is communication only between adjacent levels. In the following, we will refer to components that are directly accessed by any given component as the latter's *child components*.

There are "framework aware" activities done at all three levels: at compile-, link- and runtime.

At compile time, there is extra coding necessary that will endow the physical component, be it a library or class, with the capabilities required by the framework. The definition of "dangling bonds" and statically set bindings is up to the developer. The glue-code can be written manually or generated automatically.

At link-time, the user decides on the particular wiring to be established between the components while the framework verifies the connections and sets the

references. At this point, a variety of connectors need to be employed and configured, some manually by the user, some automatically by the framework. A typical example for manual configuration of a connector would be the case of two components that provide and require similar data but the correspondence between the parameters in the argument list is not obvious, e.g. (float pressure, float temperature) $\neq$ (float temperature, float pressure). However, the framework can set the propagator connectors by itself if the connected input and output ports are compatible. Other, special connectors fall in between these two extremes. They will require indications from the user while most of the settings are carried out by the framework.

In this approach, there is a uniform handling of calls at all levels of the hierarchy. As described in [2], it is necessary that runtime calls are done directly between the different components, that is, they are not intermediated by the framework. At the lowest level, the framework will call a component that will do all the invocations of the second level components. Following present practices this will often be a shell script or equivalent. In other cases it will be just an automatically generated propagator, such as in Figure 2.

## 3. Wiring architectures

We can distinguish between two largely different ways of managing the access of components to the references of their children:

(1) using a pipe&filter architectural style, wherein components posses only the reference data that exclusively concerns them and pass to their children the subtree of information that is necessary.
(2) using a repository-based architecture. Here a collectively accessible data repository is consulted by each component and the necessary information extracted.

Both architectures have beneficial and disadvantageous implications which we will explore in the following.

3.1. **A pipe&filter architecture.** This alternative confers more autonomy to the logical constituents of a project, i.e., to the functions. We can even say that this approach favors the logical component view on the available component base. The interfaces of all logical components will need to be extended with a new argument through which the reference tree is communicated to the component, Figure 3a. We will refer to it as *paramString* as it can be a simple XML string or an equivalent representation of it such as a data aggregate (structure or object) describing an XML node. Every function will receive a *paramString* containing child information and invokes child components by sending, along other data, a subtree of the *paramString*. An important disadvantage of this architecture is that the signature of all functions will be unnaturally "distorted" because of the *paramString* argument that has to be passed along. Moreover, the implementation

of the functions should be changed such that the parsing of the *paramString* could be done. It is feasible to parse the *paramString*s and set the references only once during the linking of the project by storing the references in internal static variables of each component.

3.2. **A repository architecture.** The suggested architecture enforces an object-oriented approach and confers a more important role to the physical aspect of components. The framework may create as many instances of a physical component (Java class, C library, etc.) as many times one of its methods are used in the project. Each instance will manage the reference data of its methods to other "child methods" internally, storing them in instance variables or global variables in case of a C library. The most important advantage is that the logical components, the methods, will not need to pass *paramString*s to each other during runtime and they will not need to have any implementation part dealing with *paramString*. Similarly to the pipe&filter architecture, the parsing of the *paramString* and setting the references is done only once per project. The beauty flaw that appears in this case is the global scope of the reference variables, which is usually not preferred by those writing computational library functions. On the other hand, existing libraries can be much more readily adapted to the framework.

In conclusion, the repository approach has the important benefit of not requiring modifications in the interfaces (signatures) of the logical components (functions). Moreover, unlike the pipe&filter architecture, the implementation of these components do not require any glue-code either. These benefits come at the cost of child reference variables with the scope of the whole physical component. In the pipe&filter case, XML parsing has to be included into all functions. There is more effort needed for implementing those indirections for each method, but some may prefer to pay this price for making the child reference variable's scope local.

Fortunately, the benefits of the two approaches can be collectd into an architecture that behaves like the pipe&filter variant during link-time and exhibits repository architecture traits during runtime.

3.3. **Mixed architecture.** There are different solutions sketched in Figure 3. Even though the last three show traits of the repository architecture, they are rather variants of the pipe&filter version. Each of them do a better or worse job in avoiding the three main problems that characterize this architecture: (i) the necessity of modifying the body of the function to include *paramString* parsing and the initialization of children references, (ii) the modified function signature that is used when being called and/or when calling the children, and (iii) the indirection introduced within the body of the physical component.

In Figure 3a, the signature includes an extra *paramString* entry and the body of the function is also modified.

Solution shown in Figure 3b moves the wiring into a separate function. In this way, the actual function can retain its most natural form both from the point of view of the interface and of the implementation. The drawback is the indirection that is introduced. Another imperfection is the fact that outgoing calls, i.e., calls to children will still require an outgoing interface that includes the *paramString*.

The third alternative, illustrated in Figure 3c, eliminates this problem by offering two different entry points at link-time and at runtime. However, this comes at the cost of an extra level of indirection. In this setup, function calls only require the passing of actual data.

One can eliminate the second level of indirection during runtime, the way it is depicted in Figure 3d. The runtime entry point is called once also during link-time and the children references are requested from the function in charge with wiring. As a result, the wiring function can be completely avoided during runtime. The actual business logic will be in the body of a function that receives all necessary information, including the child references, as parameters. This has the additional benefit of self-containment. The function is fully functional also outside the context of the framework without confusing extra arguments. On the other hand, there are two other functions containing glue-code of no interest for the component developer. However, the burden of writing glue-code can be taken over by auxiliary applications once the interface descriptor is available.

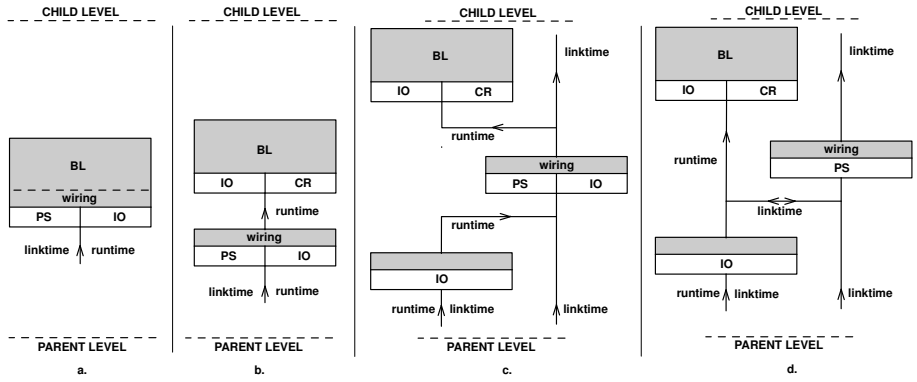


FIGURE 3. Different binding solutions for the pipe&filter architecture. Notation: PS = *paramString*, IO = input/output, BL = business logic, CR = children references

3.4. **Putting it all together.** At times, the different components will share a small amount of data, sometimes large chunks of it. The exchange may happen once in a while or frequently. As long as small amounts of data is flowing occasionally from one component to the other any binding will do it. If the amount

is large or the communication is frequent or both, performance and storage issues should be considered.

In Figure 1 we saw two different communication types: framework-component and component-component. Since the framework-component connection is done dynamically through reflection, the whole communication process imposes a much larger overhead than direct connection between components. In C, sharing large amounts of data is done in a straightforward manner as methods pass each other pointers to the array they want to share. This has the benefit of avoiding data replication. Assuming that the framework is implemented in Java, the Java Native Interface (JNI) also passes objects by reference. However, this does not solve the problem of large arrays. They get copied to another location in the memory and upon leaving the native function, the original data gets updated with the modified one. This makes JNI in its original form unfit for the job. Fortunately, since version 1.4 of the JDK, JNI has been renewed with new features targeting exactly the problem of performance. Previously, Java introduced a new package of IO classes which performs much better than the original one. The new JNI makes use of this package and provides more low level approach to data communication between object methods. This, to some extent, solves the problem of passing large chunks of data via the framework.

Frequent data passing occurs at higher levels in the hierarchy between components which heavily use the specialized services of some lightweight child components.

When connecting components, a simple syntactic check of primitive data types is not sufficient. Data types relevant to science, such as temperature, electric resistance, particle number, represent a certain amount of semantics. The types preferred by the computer are: double, long, etc. One alternative for including semantics would be the definition of classes and therein all properties of the given quantity. This would reduce the semantic problem to a regular syntactic one. The main problem of this approach is that a consistent organization of the inheritance tree of all quantities used in natural sciences may not be feasible. Moreover, the used low level languages do not offer a natural way of dealing with objects.

As for now, the connection could be done by the user tying explicitly outputs and inputs that are semantically compatible. The syntactical validation can be done simultaneously by the framework. There are reversed situations when two entry points are compatible semantically but different data types are used. For instance, electric charge is described using integer data type in one component and floating point in another. This will require simple connector objects.

## 4. Conclusions and outlook

We have shown that there is a way to harmonize all requirements for a wiring mechanism. The suggested architecture neither causes execution overhead, nor requires extra implementation from the developer, nor distorts in any way the

signatures of functions when adapting them to COMODI. As a next step, one has to look into the actual form of the glue-code that is generated so that it could cope with tasks such as callbacks to the framework for exception and error handling, project monitoring, and user interaction management. Achieving this for the relevant languages in the spirit of Babel [5] is expected to be a considerable technical challenge but none that is insurmountable.

## 5. Acknowledgments

This research is supported by the Netherland's Organization for Scientific Research (NWO) with grant number 048.031.003 and by the National Research Council (CNCSIS) with grant code 37/2004. One of the authors (Zs.I.L) is especially indebted to Prof. Dr. Simon de Leeuw from the Physical Chemistry and Molecular Thermodynamics Group at the Department of Chemical Technologies of the Technical University of Delft, The Netherlands for sharing his own and his group's extensive experience in computer simulation. The insightful comments of dr. Jouke Heringa are gratefully acknowledged.

## References

[1] *COMODI homepage,* http://phys.ubbcluj.ro/
[2] Zs.I. Lázár and B. Pârv, *COMODI: Guidelines for a Component Based Framework for Scientific Computing*, Studia Universitatis Babes-Bolyai, Seria Informatica 49 (2), 2004, pp. 91–102
[3] 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)
[4] C. Szyperski, D. Gruntz and S. Murer, *Component Software; Beyond Object Oriented Programming*, 2nd edition, Addison-Wesley (2002)
[5] *Babel homepage,* http://www.llnl.gov/CASC/components/babel.html

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